National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

Canada

NL-339 (r.86/06)

THE UNIVERSITY OF ALBERTA

PERFORMANCE OF RESILIENT SYNCHRONIZATION MECHANISMS FOR DISTRIBUTED DATABASES

BY

TSE-MEN KOON TIT MING

©

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

SPRING, 1986

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-30282-8

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: TSE-MEN KOON TIT MING

TITLE OF THESIS: PERFORMANCE OF RESILIENT SYNCHRONIZATION MECHANISMS FOR DISTRIBUTED DATABASES

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: MASTER OF SCIENCE

YEAR THIS DEGREE GRANTED: SPRING 1986

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) M Koon Tit Ming

PERMANENT ADDRESS:

Impasse Adam

Curepipe Road

Mauritius

Date: January 31, 1986

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled PERFORMANCE OF RESILIENT SYNCHRONIZATION MECHANISMS FOR DISTRIBUTED DATABASES submitted by TSE-MEN KOON TIT MING in partial fulfilment of the requirements for the degree of MASTER OF SCIENCE.

.....

Supervisor

.....

.....

.....

Date: January 31, 1986

Abstract

This thesis is concerned with the consistency, resiliency, and performance issues in distributed databases. The consistency and resiliency problems are discussed and the various techniques that are available to handle these problems are surveyed. A performance evaluation methodology for distributed database systems based on the simulation of systems expressed as extended Petri nets is presented. The modeling formalism is described and the tool that is developed to exploit this formalism for performance evaluation purposes is explained. Performance results obtained for two popular resilient update synchronization mechanisms for distributed databases are presented and analyzed.

Acknowledgements

I wish to express my deep gratitude and appreciation to my supervisor, Dr. M. Tamer Ozsu, whose patience, guidance, encouragement, and advice have made this thesis possible. I am especially grateful to him for trusting me with his invaluable collection of books and research papers and for providing a very friendly environment for research.

I would like to thank the members of my examining committee, Dr. W.W. Armstrong, Dr. D.A. Szafron, and Dr. J. Mowchenko, for their useful suggestions and comments.

Thanks are also due to David Meechan for his useful criticisms on an initial draft of some of the chapters.

I am very grateful to the members of my family, especially my two sisters, Yam and Choye, for their unselfish and devoted support, both spiritually and financially.

Table of Contents

Chapter	Page
Ch. 1: Introduction	1
1.1 What is a Distributed Database?	2
1.2 Benefits of DDBs	2
1.3 Open Problems in DDBMS	4
1.4 Need for Performance Analysis Work	5
1.5 Search for Better Modeling Tools	6
1.6 Thesis Objectives and Plan	8
Ch. 2: Concurrency Control and Resiliency in Distributed Databases	10
2.1 Terminology	10
2.2 Distributed Database Consistency Problems	11
2.3 Internal Consistency Mechanisms	14
2.4 Mutual Consistency Mechanisms	15
2.4.1 Centralized Locking	15
2.4.2 Distributed Locking	16

2.5 Deadlock 16

2.6 Resiliency 18

2.6.1 Types of Failures 19

2.6.2 Site Failure Detection 20

2.6.3 Redo/Undo Logs 20

2.6.4 Two-Phase Commit Protocols 21

2.6.5 System Reconfiguration and Transactions Termination 24

2.7 Site Recovery 25

Ch. 3: Petri Nets and Extended Place/Transition Nets 26

3.1 Petri Nets 26

3.2 Modeling of Systems Using Petri Nets 29

3.3 Extended Place/Transition Nets 30

3.3.1 Precondition 32

3.3.2 Transition Time 33

3.3.3 Transition Procedure 33

3.3.4 Output Resolution 33

3.4 Modeling of Systems Using EPTN 34

3.4.1 Terminal Site Model 35

3.4.2 Processor Site Model 36

3.4.3 Network Model 37

Ch. 4: SIMNET: An EPTN Model Simulator 41

4.1 Design Objectives 42

4.2 A Short Introduction to SIMULA 43

4.2.1 Class 43

4.2.2 Prefixed Class 44
4.2.3 Context 45
4.3 Programming in SIMNET 45
4.3.1 Program Structure 45
4.3.2 Performance Model and Assumptions 46
4.3.3 Declaration of Global constants 46
4.3.4 Declaration of Global Variables 47
4.3.5 Declaration of Data Objects 48
4.3.6 Declaration of Additional Classes and Procedures 48
4.3.7 Declaration of Transitions 49
4.3.8 Declaration of Subnets 51
4.3.9 Net Creation 51
4.3.10 Simulation Control Statements 52
Ch. 5: The algorithms 55
5.1 Assumptions 55
5.2 The Resilient Centralized Locking Algorithm 57
5.2.1 Site Model for CL algorithm 58
5.2.2 Network Model for CL Algorithm 62
5.2.3 Failure Considerations 63
5.2.4 Recovery Considerations 65
5.3 The Resilient Distributed Locking Algorithm 65
5.3.1 Site Model for DL algorithm 67
5.3.2 Network Model for DL algorithm 70
5.3.3 Failure Considerations 71

5.3.4 Recovery Considerations 71

Ch. 6: Simulation Model 72

6.1 Simulation Models 72

6.2 Input Parameters 74

6.3 Performance Metrics 76

6.4 Conflict Resolution 76

Ch. 7: Simulation Results 78

7.1 Effect of Interarrival Time 78

7.2 Effect of Base-set Size 81

7.3 Effect of Number of Sites 83

7.4 Effect of Transmission Time 84

7.5 Effect of I/O Synchronization Time 85

7.6 Effect of Database Size 88

7.7 Effect of Mean Time Between Failures 89

7.8 EPTN Simulation Versus Discrete Event Simulation 91

Ch. 8: Conclusion 95

8.1 Suggestions for Future Research 97

Bibliography 98

Appendix 1: SIMNET User's Manual 104

Appendix 2: SIMNET Implementation of On-Line Information System 131

Appendix 3: Centralized Locking Algorithm 136

Appendix 4: Distributed Locking Algorithm 139

Appendix 5: Performance Results 141

List of Figures

Figure	Page
2.1 A deadlock situation in a database system	17
2.2 Two-phase commit protocol	22
3.1 An example of a Petri net	27
3.2 Representation of concurrency in a Petri net	28
3.3 Representation of conflict in a Petri net	28
3.4 A Petri net model of a simple computer system	30
3.5 Terminal site model	36
3.6 Processor site model	37
3.7 Interpretations for On-line Information System Model	39
3.8 EPTN model of On-line Information System	40
5.1 Site model for CL algorithm	60
5.2 Network model for CL algorithm	62
5.3 DL algorithm site model	69
5.4 Network model for DL algorithm	70

7.1 Effect of mean interarrival time on mean response time 79

7.2 Effect of mean interarrival time on I/O utilization 80

7.3 Effect of mean interarrival time on CPU utilization 80

7.4 Effect of mean interarrival time on mean number of messages 81

7.5 Effect of mean base-set size on mean response time 82

7.6 Effect of mean base-set size on I/O utilization 82

7.7 Effect of number of sites on mean response time 83

7.8 Effect of number of sites on I/O utilization 84

7.9 Effect of transmission time on mean response time 85

7.10 Effect of I/O synchronization time on mean response time 86

7.11 Effect of I/O synchronization time on I/O utilization 87

7.12 Effect of interarrival time on response time (IOs = 0) 87

7.13 Effect of interarrival time on I/O utilization (IOs = 0) 88

7.14 Effect of database size on mean response time 89

7.15 Effect of mean time between failures on response time 90

7.16 Effect of mean time between failures on I/O utilization 90

7.1 Effect of mean time between failures on availability 91

A1.1 An EPTN model of a simple transition 114

A1.2 An EPTN model of a subnet 116

A1.3 An EPTN model of a net 119

List of Tables

Table	Page
5.1 Input parameter values	75
7.2 Comparison of EPTN and DES. Effect of interarrival time on mean response time	91
7.3 Comparison of EPTN and DES. Effect of interarrival time on I/O utilization for CL algorithm	91
7.4 Comparison of EPTN and DES. Effect of interarrival time on I/O utilization for DL algorithm	91
A5.1 Effect of interarrival time on response time	142
A5.2 Effect of mean interarrival time on I/O utilization	142
A5.3 Effect of mean interarrival time on CPU utilization	143
A5.4 Effect of mean interarrival time on mean number of messages	143
A5.5 Effect of mean base-set size on mean response time	144
A5.6 Effect of mean base-set size on I/O utilization	144
A5.7 Effect of number of sites on mean response time	145
A5.8 Effect of number of sites on I/O utilization	145

A5.9 Effect of transmission time on mean response time 145

A5.10 Effect of I/O synchronization time on mean response time 146

A5.11 Effect of I/O synchronization time on I/O utilization 146

A5.12 Effect of interarrival time on mean response time (IOs = 0) 146

A5.13 Effect of mean interarrival time on I/O utilization (IOs = 0) 147

A5.14 Effect of database size on mean response time 147

A5.15 Effect of interarrival time on response time for different MTBFs 148

A5.16 Effect of mean interarrival time on I/O utilization for different MTBFs 148

Chapter 1

Introduction

Distributed database management systems (DDBMS), that is, the organizational structure of distributed data and the support software, have been the focus of intensive research during the past few years. If we follow the trend, we can easily foresee that their importance will rapidly grow. There are several technological and organizational reasons for this trend: (1) recent advances in microelectronics technology have made computer hardware affordable that previously was too expensive to be duplicated; (2) increased user demands for information have strengthened the need for faster information retrieval systems; (3) developments in computer networks have made computer communications cost effective and efficient; (4) advances in database technology have provided a solid foundation for the development of distributed databases; and (5) distributed databases seem to fit more naturally to today's large decentralized types of organizations [CeP84].

In spite of the fact that the field of DDBMS is still in its infancy and that there is yet no well established methodology that could be used for the development of distributed database systems, some large corporations have already invested large amounts of money in and are relying heavily on large databases distributed over worldwide

networks of computers [Gra78], and [Che81]. Due to the vast amount of data that are usually stored, retrieved, processed, and transferred on these systems and the dependence of the institutions on the accuracy, availability and promptness of the data, the system must on the one hand be reliable and efficient and on the other hand ensure the consistency of the data. This thesis is concerned with the problems of *consistency*, *reliability*, and *performance* of distributed database systems.

1.1. What is a Distributed Database?

Since the field of distributed databases is relatively new, it is usually given different meanings and defined differently by different researchers. Thus, in order to have a common frame for discussion, we will first attempt to define and explain what we mean by a distributed database. A *distributed database* (DDB) is defined as a collection of logically interrelated data items distributed over two or more computer sites interconnected by a network [Ozs82]. There are two elements of significance in the above definition. First, the sites are geographically separated and can communicate with each other only over some communication network. This is a necessary condition since by having a network as the only medium of communication another level of complexity is added to the problems of data distribution that otherwise might not be present. These problems are discussed in Section 1.3 and Chapter 2. Second, the data should have a certain amount of logical relationship which ties them together. This distinguishes a distributed database from a set of local databases or files located at different sites of a computer network.

1.2. Benefits of DDBs

Besides technological and organizational reasons, there are several other factors that have motivated the development of distributed databases. These are the

numerous potential benefits that they offer over the conventional centralized databases. The most commonly mentioned benefits are:

1. **Reliability and Availability** - By having duplicate data at several sites, higher reliability and availability can be achieved. Failure of any one site does not necessarily cause a total system failure since the data can still be accessed from another site. Even in cases where a combination of site crashes causes part of the data to be inaccessible, the operational sites can still provide limited service.

2. **Improved Performance** - With the distributed system, it is possible to decompose the database and store portions at locations where they are most frequently used, thus reducing the time to access the data. Performance can also be improved by taking advantage of the inherent parallelism, for example, by decomposing queries so that searches can be performed concurrently at different sites.

3. **Local Autonomy and Security** - By partitioning the database, each site can be given entire control over its own data while being able to share it with other users. This results in better security, privacy and management of the data.

4. **Economy** - In a geographically distributed database where the type of applications is highly localized, a lot of the processing can be done locally, thus reducing the communication cost usually involved in accessing a remote central database.

5. **Expansibility** - In a distributed environment, it is much easier to make incremental changes with a minimum impact on the already existing units.

6. **Sharability** - In addition to the ability to share data among the sites that are interconnected, it also possible to share expensive peripheral devices.

1.3. Open Problems in DDBMS

However, the benefits mentioned above are only potential benefits, and before they could be fully realized several problems need to be overcome. These problems include query processing, database design, directory management, deadlock, concurrency control, and resiliency.

1. **Query Processing** - The objective is to develop algorithms that will attempt to optimize the processing of queries taking into consideration the inherent parallelism of distributed systems, the transmission delay, the communication cost, etc. Various algorithms are discussed in [SaY82], and [AHY83].

2. **Database Design** - The research in this area essentially involves the use of operations research techniques to determine the optimal placement of data under different constraints. A review of the work in this area can be found [ChA80].

3. **Directory Management** - This problem is related to the database design problem in the sense that the same techniques are applied to determine the optimal placement of directories. These problems are discussed in [ChN75]

4. **Deadlock** - This problem is similar in nature to that encountered in operating systems. The research in this area is concerned mainly with the development of mechanisms for the prevention, avoidance, detection and resolution of deadlocks in distributed database systems ([ChO74], [Mal80], and [Obe82]).

5. **Concurrency Control** - Concurrency control is concerned with the synchronization of concurrent accesses to a distributed database so that the consistency of the database is preserved. Most of the research effort in this area, so far, has been geared towards the design of new algorithms. Only recently has there been a small shift towards the analysis and performance of the algorithms. A survey and an analysis of this problem could be found in [BeG81].

6. **Resiliency** - This problem is also known as the *crash recovery* problem. In a distributed system with a large number of independent components, there is always the chance that one or more of the components may fail. Whenever a failure occurs it is very important that the database consistency is not violated. In order to achieve this, the system must be resilient, that is, appropriate mechanisms should be provided to allow the system to detect and recover from failures. Furthermore, it is desirable that the system allows some query and update activity to occur at the operational sites during a failure. Most of the work in this area is concerned with the design of procedures that would allow partial operability of the system while preserving the consistency of the database in the presence of failures, detect errors and failures, prepare sites for recovery, and recover a site from a failure. The crash recovery techniques are surveyed in [Koh81].

1.4. Need for Performance Analysis Work

Undoubtedly, the problem that has received the most attention during the last few years is concurrency control. Unfortunately, most of the work has concentrated on the development of new algorithms ([Ell77], [Lan78], [BaP78] and [Tho79]), and not as much attention has been paid to performance analysis. From the current literature, one can name dozens of such algorithms that have been proposed, most of them with unproven workability and performance (in fact, it has been claimed that many of them are incorrect [BeG81]). Thus, what is needed now is some performance analysis work that would analyze and compare the performance of these algorithms. Some researchers have already taken first steps towards this goal, but there is still much work to be done.

The few performance related work on concurrency control algorithms includes those by Garcia-Molina [Gar79], Ries [Rie79], Cheng [Che81], Lin and Nolte [LiN83],

and Ozsu [Ozs85b]. Garcia-Molina has developed analytic models for three locking-based algorithms — the centralized scheme, the voting scheme, and the ring scheme — and verified the results using simulation. Ries has developed simulation studies primarily aimed at finding the effect of locking granularity on system performance. Cheng has used Garcia-Molina's analytical methodology to analyze various resilient algorithms. Lin and Nolte have studied the relationship between the read/write ratio of transactions and system performance. Ozsu has developed and used an *Extended Place/Transition Net (EPTN)* formalism, a derivative of Petri nets, to model and analyze some two-phase locking algorithms.

In the area of distributed databases, the problems of concurrency control, deadlock resolution, and resiliency are strongly interrelated. For example, a good concurrency control algorithm, in addition to being efficient in synchronizing update transaction, must provide satisfactory (if not optimum) performance in the areas of deadlock resolution and resiliency. However, due to the complexity of the problem as a whole, most of the performance analysis work, with the exception of Cheng's work, is based on the simplifying assumption that system failures do not occur. This allows the crash recovery problem to be ignored. The major criticism against this approach is that oversimplification may lead to unrealistic results. Thus, what is also needed in the area of performance analysis, is a study of the performance of concurrency control, deadlock resolution, and resiliency mechanisms as an integrated problem.

1.5. Search for Better Modeling Tools

One of the main reasons for the lack of performance related work in this area might be the unavailability of good modeling tools. Performance analysis based on queuing theory, which is quite adequate for simple systems, becomes too complicated to handle as the systems become more complex. Discrete event simulation methods,

although providing an alternative to queuing analysis when the system is too complex, are only appropriate for conventional sequential systems, and are not adequate for today's complex concurrent systems [Age78]. Some researchers have delved into the problem hoping to come up with a suitable modeling tool. The tool would lend itself easily to the performance evaluation and possibly verification of these complex systems. One model that is being considered is Petri nets.

Petri nets, also called Place/Transition nets, have evolved from the work of Carl Adam Petri [Pet62], A.W. Holt [HoC70] and many others, and were found to be particularly suitable for representing and studying systems that contain asynchronous and concurrent activities. Some of the commonly mentioned features of Petri nets are [Age78]: (1) the graphical and precise nature of the representation scheme often makes the overall system easier to understand and to communicate to other people, (2) Petri nets are analytically powerful and a growing body of knowledge exists on the use of Petri nets theory for analyzing the behavior of systems, and (3) due to the structuredness of the scheme, Petri nets can be synthesized using either bottom-up or top-down approaches, thus, making it possible to systematically design systems from known or easily verifiable sub-systems.

However, as a tool for the performance modeling of systems, Petri nets have many shortcomings. These will be discussed in Chapter 3, after an introduction to Petri nets. Many attempts have been made to solve these problems, and these have resulted in various Petri net derivatives such as Evaluation Nets [Nut72], Pro-nets [Noe79], Predicate/Transition nets [GeL79], and Extended Place/Transition nets (EPTN) [Ozs85a]. Among these derivatives, the EPTN formalism has been developed particularly for the modeling and analysis of distributed database systems, although it is also suitable for distributed systems in general.

1.6. Thesis Objectives and Plan

The primary objective of this thesis is to study the performance of resilient locking-based concurrency control algorithms, namely, the *centralized locking* and *distributed locking* algorithms, in an environment where site failures could occur. The study is carried out using two different approaches. In the first approach, discrete event simulation (DES) is used to obtain performance results. In the second approach, a new methodology based on the simulation of systems modeled as EPTN is used for the performance analysis. In doing so, we hope (1) to show the appropriateness of the EPTN formalism as a performance modeling tool for asynchronous and concurrent systems, (2) to uncover some of the underlying factors that should be taken into consideration when designing a resilient distributed concurrency distributed concurrency control algorithm, (3) to draw some useful conclusions regarding the tradeoffs that are involved in selecting a concurrency control algorithm for a distributed database, and (4) to clear up some of the controversies that have surrounded these two algorithms in the past.

In order to facilitate the implementation and simulation of systems expressed as extended Petri nets, a simulation package SIMNET, hosted by the simulation language SIMULA ([DaN66], [Fra77], [Bir81a], [Lam83], and [DMN84]) is developed. SIMNET is designed so that it could be used to implement and simulate not only concurrency control algorithms for distributed systems but also any type of systems that can be expressed as an extended Petri net.

This thesis is organized as follows. In chapter 2, we present a review of the background material on distributed databases and introduce the basic techniques that are used for concurrency control and resiliency. In chapter 3, we present an overview of Petri nets and the extended Petri net formalism. In Chapter 4, SIMNET is introduced. Chapter 5 presents the two algorithms that are studied along with the EPTN models

for the algorithms. Chapter 6 describes the simulation model, the input parameters to the model, and the performance metrics that are used. In Chapter 7, the simulation results are presented together with an analysis and discussion of the results. Finally, in Chapter 8, we give a summary of the results obtained from this study, and make some suggestions for further research.

Chapter 2

Concurrency Control and Resiliency in Distributed Database Systems

This chapter introduces some of the problems that are commonly encountered in designing resilient distributed database systems and surveys several of the techniques that are commonly used to handle these problems. Among the topics that are discussed are concurrency control, deadlock resolution, and resiliency. At the same time, the database model for this study is introduced.

2.1. Terminology

In a distributed database, each data item may be stored at any site in the system or stored redundantly at several sites. A distributed database is said to be *partitioned* if there are no duplicate items; *partially replicated* if part of the data is duplicated; and *fully replicated* if the entire database is duplicated at all sites. In this study, we are concerned mainly with fully replicated databases.

A distributed database is said to be *consistent* if all replicated portions of the database are both *internally consistent* and *mutually consistent*. Internal consistency, which is fundamental to both centralized and distributed database systems, implies

that the entity values in any copy of the database satisfy a set of *integrity constraints*. Integrity constraints refer to *correctness assertions* that are associated with the values of every data item in the database. An example of such assertions is: 'The salaries of employees in department x should be between 2,000 dollars and 80,000 dollars'. Mutual consistency, which is specific to distributed database systems, means that all the values of multiple copies of any data item converge to the same final value should the system stop receiving new transactions.

A user interacts with a database by means of *transactions*. A transaction is defined as a sequence of primitive atomic operations, for example, reads, computes and writes, that maps the database from a consistent state to another consistent state [Lam78]. A transaction is thus a larger unit of atomic action on the database state.

Transactions could be categorized into two types: *read-only* transactions and *update* transactions. In this thesis, we are interested mainly in update transactions. An update transaction is assumed to consist of the following steps:

1. **Read** - the items needed by the transaction are read.
2. **Compute** - new values are computed.
3. **Write** - the distributed database is updated, that is, all duplicate items that need to be modified are updated to reflect the new values.

Two transactions are *conflicting* if they operate on the same data item and one of them is a write.

2.2. Distributed Database Consistency Problems

In a distributed database, there are basically two types of problems: (1) the internal consistency problem, and (2) the mutual consistency problem.

Internal Consistency Problem: The internal consistency problem arises when several conflicting transactions attempt to modify a copy of the database at the same time. This situation is illustrated by the following example. Consider only one copy of a distributed database and the following two conflicting transactions:

```
T1 : Read x;                    T2 : Read x;
     x1 := x + a;                    x2 := x - b;
     Write x1 into x;                Write x2 into x;
```

If the two transactions are executed concurrently, it is possible for the transactions to read the data item x, compute the new value for x, and then store the new value into the database at approximately the same time, as shown below.

```
T1 : Read x;
T1 : x1 := x + a;
T2 : Read x;
T2 : x2 := x - b;
T1 : Write x1 into x;
T2 : Write x2 into x;
```

If this happens, the final value of x is incorrect, since the effect of one transaction is overwritten by the other transaction. In the example above, the effect of T1 on x is lost and the final value of x is $x-b$ instead of $x+a-b$ or $x-b+a$, as one would expect. This is known as the *lost update anomaly* and is one of the several types of internal inconsistencies [BeG81] that could occur in both the centralized and distributed database systems.

Mutual Consistency Problem: The mutual consistency problem occurs in fully or partially replicated databases because of the requirement that copies of the database must be identical. In order to keep all copies identical each update transaction must be applied uniformly and simultaneously to every copy of the database. However, because of communication delays and failures, it could happen that updates are

applied to different copies at different times and in different orders. If this occurs and the updates are not controlled, the mutual consistency of the database could be affected. This situation is illustrated in the following example. Consider two sites A and B, geographically separated and linked together by a network, each one with a duplicate copy of the database. Assume that two transactions are received by site A and that the transactions are similar to T1 and T2 above except that before a new value is written into a copy of the database, the value is sent to the other site to be stored in the other copy as well. Now, although we assume that an internal consistency check is performed and T1 and T2 are executed sequentially to prevent lost updates, the consistency of the distributed database is still not guaranteed. Inconsistency could occur if the messages sent by one site are not received and processed in the same order by the other site. Note that in a distributed system, there is no guarantee that the messages will be received in the same order that they were sent. This is illustrated below:

Site A	**Site B**
T1 : Read x;	Receive x2 from site A;
x1 := x + a;	Write x2 into x;
Send x1 to site B;	Receive x1 from site A;
Write x1 into x;	Write x1 into x;
T2 : Read x;	
x2 := x - a;	
Send x2 to site B;	
Write x2 into x;	
Final value of x : x+a-b	Final value of x : x+a

When this occurs the final state of the database is inconsistent since the transactions are not executed in the same order at the two sites and the value of a more recent update is overwritten by an older update. In the example above, the final value of x at site B is $x+a$ instead of $x+a-b$ which is the final value of x at site A.

2.3. Internal Consistency Mechanisms

One obvious solution to the internal consistency problem is to run the transactions serially, that is, one at a time in any order. Since a transaction is a unit of consistency, any sequence of transactions executed serially without interference from other transactions also preserves consistency of the database. However, this is not a good solution since there are transactions, for example, those that do not access the same data items, that can be executed in parallel or concurrently without affecting the database consistency. Therefore, what would be more appropriate is a mechanism that would allow transactions to be executed concurrently without violating the consistency of the database. In formal terms, the mechanism would ensure that the execution of concurrent transactions is *serializable* ([BeG81], [Pap79]); that is, even though the transactions are executed concurrently the overall effect on the database is equivalent to what would have resulted if the transactions were executed in some serial order. Such a mechanism is provided by concurrency control algorithms. Many algorithms have been proposed in the literature, and two mechanisms that are most commonly used are *two-phase locking* and *timestamp ordering*. In this paper, we will be concerned mainly with two-phase locking algorithms.

Two-phase locking achieves serializability by using locks to isolate conflicting transactions from each other and by requiring that locking and unlocking of items be done in two phases, known as the *growing phase* and the *shrinking phase*. During the growing phase the transaction can only request locks, and during the shrinking phase the transaction can only release locks and cannot request any additional locks. It has formally been proven that two-phase locking is a correct concurrency control method [CeP84].

2.4. Mutual Consistency Mechanisms

The problem of maintaining the mutual consistency of distributed databases has also received a lot of attention in the past few years, and an abundance of algorithms exists in the literature. One common solution is to modify the 2-phase locking mechanism so that it preserves the mutual consistency of the distributed databases as well. The two basic strategies that are usually incorporated with the mechanism are known as: *centralized locking* and *distributed locking*. The two algorithms studied in this thesis are actual implementations of these two strategies.

2.4.1. Centralized Locking

In this strategy, one of the sites, called the central site, is chosen *a priori* (using an election protocol [Gar79], [see Section 5.4 also]) to control the allocation and deallocation of locks to transactions. Before a transaction can access data at any site the appropriate locks must be requested and obtained from the central site. For example, if a transaction wants to update, let's say, an item x, the site where the transaction originates must send a lock request for x to the central site, wait for the lock granted message from the central site, and then proceed with the update of item x. After the update is completed at all the sites, a message to unlock the item is sent to the central site. This mechanism ensures a total ordering among conflicting transactions so that both internal and mutual consistencies are preserved.

The common criticisms against this strategy are: (1) the central site could be a performance bottleneck since all the update transactions must visit the site to obtain the locks, and (2) the reliability of the entire system is too dependent on one site: a central site failure causes a total system failure. Some approaches to improve the reliability of the scheme have been suggested. Alsberg and Day [AlD76] have shown that by keeping a backup of the central site, any desirable level of reliability can be achieved.

However, Cheng [Che81] has carried out performance studies and has found that the performance of the resulting scheme is affected considerably due to the additional overhead incurred. Garcia-Molina [Gar79] has suggested an alternative approach that uses election protocols to rapidly recover from a central site failure, but no studies have yet been carried out to find how this scheme would perform. In this thesis, we will attempt to study this scheme and we hope to draw some definite conclusions regarding its performance.

2.4.2. Distributed Locking

Instead of having the lock management duties performed at only one site, these duties could be performed at every site in the system. However, in this strategy, before a transaction could update a data item at any site, locks for the items must be requested and obtained from every site in the system. Similarly, after the completion of the update at every site, the locks for the item should be released at every site. The major difference between this strategy and the centralized locking strategy is that, in this strategy, a lock table is kept at every site so that: (1) the read-write or write-read conflicts between transactions could be controlled locally, and, as a result, only write locks need to be requested from the other sites; and (2) failure of any site does not cause a total system failure since the lock table could still be accessed from any operational site.

The distributed locking strategy has been criticized mainly because of the complexity of the algorithm and the extra overhead that is incurred during normal operation.

2.5. Deadlock

One problem with locking-based mechanisms is that they are subject to deadlock and therefore need deadlock resolution mechanisms. A *deadlock* is said to have

occurred in a database system when a transaction t_i is holding a lock for an item d_i and is waiting for an item d_j locked by a transaction t_j which is directly or indirectly waiting for item d_i. A simple example of a deadlock is illustrated in Figure 2.1. An arrow from a transaction, depicted as a rectangle, to an item, shown as a circle, indicates that the transaction is waiting for the item. An arrow from an item to a transaction indicates that the item is locked by the transaction.

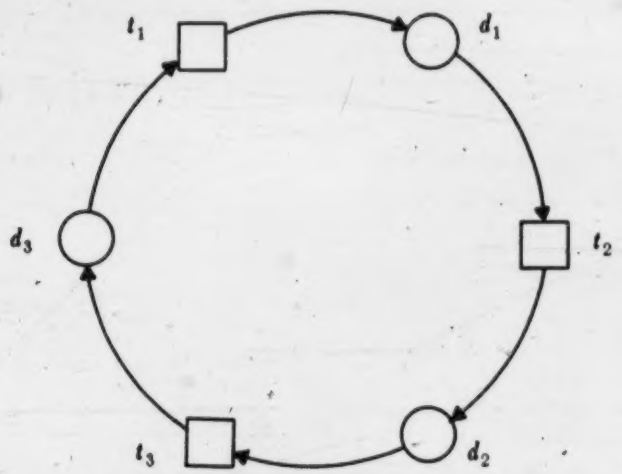

Figure 2.1. A deadlock situation in a database system.

There are three schemes that are commonly used for handling deadlock. These include *deadlock prevention*, *deadlock avoidance*, and *deadlock detection*.

In the deadlock prevention scheme, if a transaction t_1 requests a resource that is already held by another transaction t_2, a deadlock "prevention test" is run. If the test indicates that there is a possibility of deadlock, either t_1 is canceled (*non−preemptive*) or t_2 is aborted (*preemptive*).

In the deadlock avoidance scheme, transactions are required to request their resources in some predefined order, and a transaction can only wait for an item which

is held by an older transaction (or one with a higher priority). In this scheme deadlocks cannot occur. The algorithms studied in this thesis are based on this scheme.

In the deadlock detection scheme, deadlocks are detected and resolved after they occur. Two mechanisms could be used to detect a deadlock: *timeout* and *Wait-For-Graph* (WFG). If the timeout mechanism is used, a deadlock is assumed to have occurred if after a timeout period a transaction is still waiting for an item. When this situation occurs, the transaction is aborted. This mechanism can cause unnecessary aborts. A WFG depicts the waiting sequence of transactions for access to a data item. If the WFG mechanism is used, deadlocks are detected by searching for cycles in the graph and then resolved by aborting one or more of the transactions involved to break the cycles.

In a centralized database system, deadlocks could easily be detected since all the information about the WFG is located at one place. However, in a distributed database system, deadlock detection is much harder since the information is dispersed among the sites and, furthermore, deadlocks can occur not only locally (that is, involving only one site) but also globally (that is, involving more than one site). Detection and resolution of deadlocks in distributed systems are discussed in [Mal80] and [Obe82].

2.6. Resiliency

One of the commonly mentioned benefits of distributed database systems is its reliability/availability. However, in order for this benefit to be achieved, the system must be *resilient*, that is, the system must be able to operate correctly even in the presence of failures. In the following sections, we will look at some types of failures that could occur in a distributed database system, how they could be detected, and

how resiliency against some of them could be achieved.

2.6.1. Types of Failures

There are several types of failures that could occur in a distributed system [Gar80]. Four main categories are: (1) communication failures, (2) site failures, (3) transaction failures, and (4) network partitioning.

Communication failures can occur because of lost/duplicate messages or link failures. This category of failures is the simplest and does not pose any problem at the DBMS level. Lost messages can easily be replaced by repetitive transmission of messages until an acknowledgment is received. Link failures could be bypassed by rerouting messages. However, this might not be possible if the network is partitioned or the type of link is a bus.

The second category of failures, site failures, can occur because of a serious system error at a site, for example, processor malfunctioning or memory failure. This type of failure can affect the consistency of the database and is dealt with in the upcoming sections.

A transaction failure occurs when a transaction has to be aborted because of incorrect computations, invalid access requests, or deadlocks. Transaction failures could be handled by keeping enough information to allow the transaction to be undone or redone at a later time.

Network partitioning is a combination of link failures that splits the network into two or more subnetworks such that sites in the same subnetwork can communicate with each other but sites in one subnetwork cannot communicate with sites in another subnetwork. Network partitioning is an extremely hard problem and has not been well studied in the literature. We will not deal with network partitioning in this study.

2.6.2. Site Failure Detection

Improper detection of a failure could severely impact the correct functioning of a distributed database system. Thus, it is necessary to have a mechanism that could detect a site failure when it occurs and notify the database system. Since failure detection and notification protocols usually reside in the communication subsystem, we will only briefly discuss one mechanism that is commonly used, and then assume that whenever a failure occurs, the database system is properly notified by the communication subsystem.

The mechanism that is usually used to detect that a site has failed is based on timeout and retry [Lam78]. A site s_1 detects that another site s_2 has failed if a required message is not received within a timeout period t and possibly after some retries. One problem with this mechanism is choosing the right value for t. Too small a timeout value and number of retries could lead to false failures and too large a timeout value and number of retries could lead to longer delays. In this study, we assume that a failure is correctly detected within a reasonable time period.

2.6.3. Redo/Undo Logs

If a site fails while an update is in progress, the database might be left in an inconsistent state. To protect against this type of inconsistency, Gray [Gra78] describes a technique that makes use of *undo/redo logs* and *write ahead protocols*. Before modifying a data item in nonvolatile storage, a log record containing the old and new value of the updated item is written in a log that is then forced in *stable* storage [LaS76]. Stable storage is assumed to be robust to failures and can be constructed by replicating the same information on several disks with independent failure modes. Since the log is written in a safe place before the data is actually updated, this technique guarantees that enough information will always be available about the

update to undo or redo the update completely at any time. Now if the system fails while executing an update, the recovery routine could inspect the undo/redo log at restart and undo/redo any incomplete updates.

Other methods that could be used to reconstruct a consistent state of a distributed database include differential files [SeL76], and checkpoints and before images [Asa79], [Gra78], [Ver78].

2.6.4. Two-Phase Commit Protocols

If an update transaction is not performed uniformly at all sites, the consistency of the database could be violated. This might happen, for example, if a site fails while sending update messages to the other sites. Those sites that receive the message would commit the transaction while the other sites might abort the transaction thus causing mutual inconsistency in the database. A partial solution to this problem is provided by the *two-phase commit protocol* [Gra78]. The two-phase commit protocol ensures that an update is either committed at all sites or not committed at all (aborted at all sites), even if failures occur during its execution.

In the basic two-phase commit protocol, there is a special site called the *coordinator*. The coordinator is responsible for controlling the execution of the transaction and for taking the final decision on whether to commit or abort a transaction. The other sites that participate in the execution of the transaction are called *cohort* sites or *participants*. An informal description of the protocol is as follows (see Figure 2.2).

Initially, the coordinator records the transaction update values in stable storage and sends a *prepare-to-commit* message to all cohort sites. Recall that the coordinator is notified of any site failure by the communication subsystem. When a cohort site receives a *prepare-to-commit* message, it checks if it could commit the transaction. If the transaction could be committed, the site saves the transaction's update values in

```
Coordinator: Record transaction update values in log;
             Write "prepare" in log;
             Send prepare-to-commit message to all cohort sites;
Cohort site: Receive prepare-to-commit message;
             If transaction could be committed then begin
                  Record transaction update values in log;
                  Write "ready" in log;
                  Send agree to coordinator
             end
             else begin
                  Write "abort" in log;
                  Send abort to coordinator
             end.
Coordinator: Receive replies (abort or commit) from all cohort sites;
             If at least one abort message or failure signal is received then begin
                  Write "abort" in log;
                  Send message abort to all cohort sites;
                  Cancel transaction
             end
             else begin
                  Write "commit" in log;
                  Send message commit to all sites;
                  Perform update on local database;
             end.
Cohort site: Receive commit or abort message from coordinator;
             If commit message is received then begin
                  Write "commit" in log;
                  Perform update on local database;
                  Write "complete" in log;
                  Send acknowledgment to coordinator;
             end
             else begin
                  Write "abort" message in log;
                  Cancel transaction
                  Write "complete" in log;
                  Send acknowledgment to coordinator;
             end.
Coordinator: Receive acknowledgments from all cohort sites;
             If a failure signal is received then
                  Save information for failed site;
             Write complete in log;
```

Figure 2.2. Two-phase commit protocol.

stable memory and sends an *agree* message to the coordinator; otherwise, the site sends an *abort* message to the coordinator. After the coordinator has received a reply from every operational site, it decides on whether to commit or abort the transaction. If at

least one site has decided to abort the transaction or a failure signal is received, the coordinator sends an *abort* message to all sites and cancels the transaction locally; otherwise, it sends a *commit* message to all sites and updates the local database. At the cohort sites, if an *abort* message is received, the transaction is canceled; if a *commit* message is received, the local database is updated and the transaction is terminated. In either case, an acknowledgment is sent to the coordinator. When the coordinator receives the acknowledgments from all the cohort sites, it completes the transaction locally. If a site has failed, the decision is saved for the failed site.

The behavior of the protocol in the presence of failures is as follows:

1. If a cohort site fails before the ready record is written in stable storage, the coordinator will be notified about the failure and the transaction will be aborted at all sites. At restart the failed site could safely abort the transaction.

2. If a cohort site fails after the ready record is written in stable storage, the recovery routines should inquire about the outcome of the transaction at restart and then perform the appropriate action.

3. If the coordinator fails after the prepare record is written in stable storage, but before the commit or abort record is written, the recovery routines at restart should restart the transaction from the beginning.

4. If the coordinator fails after the commit or abort record is written in stable storage, the recovery routines at restart should send the decision again to ensure that the decision is received by all sites.

5. If the coordinator fails after the complete record is written in stable storage, no further action needs to be taken since the transaction has already been completed.

2.6.5. System Reconfiguration and Transactions Termination

One problem with the two-phase commit protocol is that it is a *blocking* protocol, that is, if the coordinator fails after sending a prepare message, the operational sites have to wait for the failure to be repaired before they could terminate the pending transaction. This could be very inconvenient since other transactions that require access to the items held by the blocking transaction will also have to wait. In some situations, however, it is possible to design a termination protocol to terminate pending transactions. The protocol would behave as follows:

1. If at least one of the cohort sites has received the decision, the other cohort sites would be told of the decision and could terminate the transaction.
2. If none of the cohort sites has received the decision, and only the coordinator site has crashed, the cohort sites could regroup and elect a new coordinator to terminate the pending transaction.

In both of the above situations, the transaction would be terminated correctly at all operational sites. However, if none of the cohort sites has received the decision and at least one participant has also failed, termination is impossible, since the operational sites cannot know what the site has decided and cannot take an independent decision.

To eliminate the above problem, a *three-phase commit* protocol has been proposed by Skeen [Ske81]. The protocol has been developed based on the following observation. The operational site in the two-phase commit are blocked because the participants go directly from the ready state to the abort or commit state. For this reason, even if all the operational sites have not received any decisions, the failed site might already have performed some definite action (abort or commit) that cannot be undone. This problem is solved in the three-phase commit protocol by having a *prepare-to-commit* state and a *prepare-to-abort* state between the ready and the commit and abort states,

respectively. In this case, if none of the operational sites have received the decision, they can still go ahead and abort the transaction. The failed site will also abort the transaction at restart. Notice that even if the failed site has reached the *prepare-to-commit* state the transaction could still be aborted since the final commitment has not been reached yet.

2.7. Site Recovery

When a site recovers from a failure, the database at that site should be brought up-to-date, that is, the updates that it missed while it was down should be executed. A brute force approach of achieving this is to copy the entire database from another up-to-date operational site. A more reasonable approach is to keep a log of all updates that the failed site may miss. The recovery routines at the failed site could then request and use this log to restore the consistency of its database at restart. The log could be maintained by a central site, by the site where the transaction originates, by some specially selected sites, or by every site.

An alternative approach is based on a technique called persistent communication [AlD76]. In this technique, the responsibility of remembering the missed updates is given to the communication subsystem. Once a message is accepted by the communication subsystem, the message is guaranteed to be delivered to the destination site eventually. This is usually achieved by keeping the message to be sent to a failed site in a reliable buffer at the originating site.

Chapter 3

Petri Nets and Extended Place/Transition Nets

In this chapter, we present an overview of Petri nets and the Extended Place/Transition Net (EPTN) formalism. For a more comprehensive study on these topics, the reader can refer to [Pet81] and [Ozs85a].

3.1. Petri Nets

A Petri net can be considered as a structure consisting of three components: a *set of places* (P), a *set of transitions* (T), and a *set of directed arcs* (A) which connect the transitions and the places. Pictorially, a Petri net is a *directed bipartite graph* as shown in Figure 1 (adapted from [Age78]). The places are represented by circles, transitions by bars, and tokens by small black dots inside the circles.

The *marking* of a Petri net represents the number of tokens that reside in each place of the net. Each marking can be considered as representing a state of the Petri net.

A Petri net executes by changing from one state to another according to certain *simulation rules*. The change in state is controlled by the *firing* of transitions. A

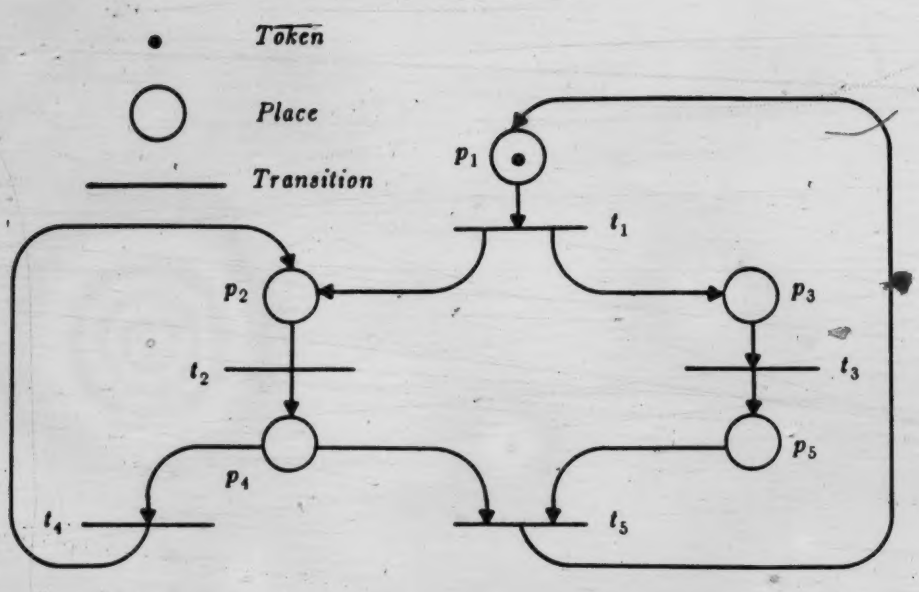

Figure 3.1. An example of a Petri net.

transition may fire only if it is *enabled*. A transition is enabled if each of its input places has at least one token in it. A transition fires by removing one token from each of its input places and depositing a token into each of its output places.

In Figure 3.1, the only transition that is enabled and can therefore fire is t_1. Transition t_1 fires by removing a token from p_1 and depositing tokens into p_2 and p_3 resulting in the new marking shown in Figure 3.2.

At this stage, both t_2 and t_3 are enabled and can fire concurrently. After the firing of both t_2 and t_3, the new marking shown in Figure 3.3 is obtained. This situation represents a *conflict*: both t_4 and t_5 are enabled and can fire, however, the firing of either one of them disables the other. The decision as to which one fires is completely arbitrary. It is this ability to easily represent both concurrency and conflict that makes Petri nets a powerful modeling tool.

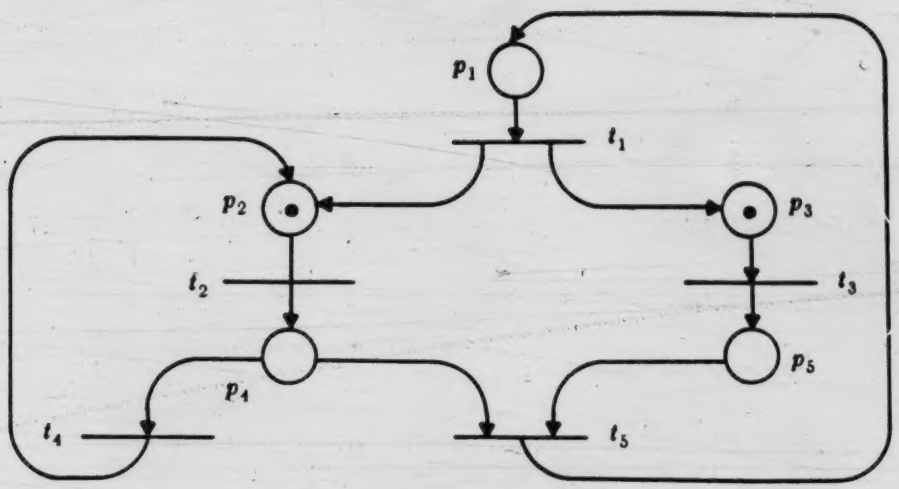

Figure 3.2. Representation of concurrency in a Petri net.

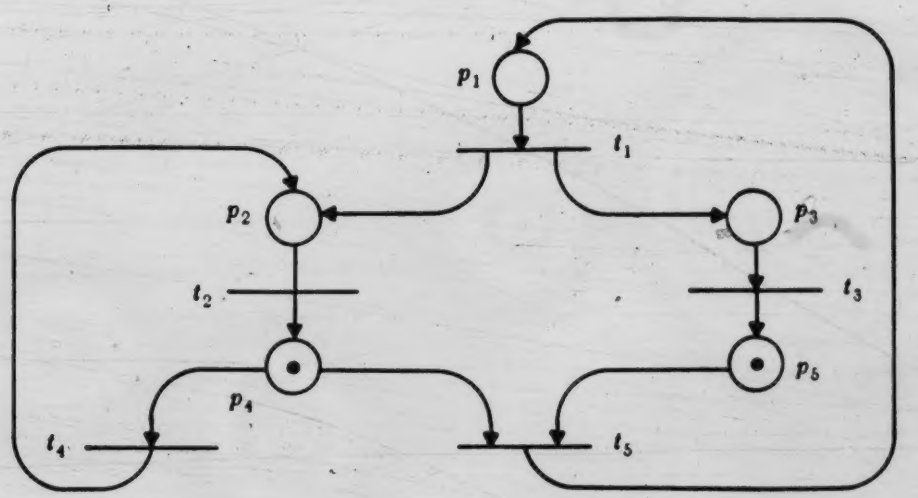

Figure 3.3. Representation of conflict in a Petri net.

3.2. Modeling of Systems Using Petri Nets

Petri nets have been specifically devised for the modeling of systems, especially systems with concurrent and parallel activities. In this section, we illustrate how to model systems using Petri nets.

In order to use Petri nets for system modeling, one must view a system in terms of *events* and *conditions*. Events represent actions that occur in the system. Conditions represent the states of the system. For an event to occur, one or more conditions must be true. These are called the *preconditions* of the event. The occurrence of an event could change the state of the system and thereby cause some other conditions to become true. These new conditions are called the *postconditions* of the event.

As an example, consider a simple computer system with one CPU. The conditions for the system are:

p_1 : a job has arrived and is waiting

p_2 : CPU is available

p_3 : job is being processed

p_4 : job is completed

The events for the system are:

t_1 : CPU starts processing of job

t_2 : CPU completes processing of job

The postcondition of event t_1 is (p_3) job is being processed. For event t_1 to occur, the preconditions p_1 and p_2 must be true, that is, for the CPU to start processing of a job, a job must be waiting and the CPU must be available. Similarly, event t_2 (CPU completes processing of job) can occur only after precondition p_3 (job is being processed) is true. The occurrence of this event will cause condition p_4 to become true.

Based on this view of the system, a Petri net model could be constructed for the system. Conditions are modeled by places in the net, events are modeled by transitions, and the holding of a condition is represented by a token in the place corresponding to the condition. The input places of a transition represent the preconditions of the corresponding event; the output places represent the postconditions. The firing of a transition is equivalent to the occurrence of the corresponding event. When a transition fires, tokens from the input places are removed thus ceasing the holding of the preconditions, and new tokens are placed in the output places causing the postconditions to become true.

The Petri net model of the simple computer system is shown in the Figure 3.4.

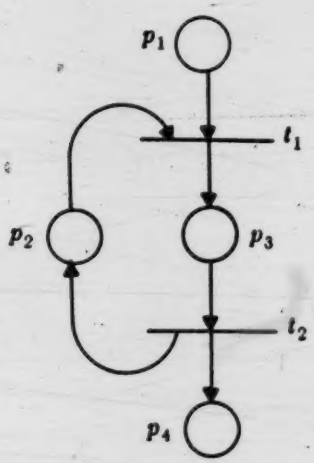

Figure 3.4. A Petri net model of a simple computer system.

3.3. Extended Place/Transition Nets

As we have mentioned in the introduction, Petri nets have many shortcomings when it comes to the performance modeling of systems. These include [Ozs85a]: (1) the lack of the time concept - in Petri nets, transition firings are instantaneous, meaning

that events do not take any time, (2) the absence of a mechanism whereby tokens which may be interpreted as representing jobs in the system are allowed to carry data, and (3) the insistence that all the input places of a transition must be full before the transition could fire; similarly, a transition firing implies that all the output places of the transition are affected by the firing.

Several derivatives of Petri nets that attempt to solve these problems have been proposed in the literature, and the one that is particularly suitable for distributed database systems is the EPTN formalism. In this section, we briefly describe a subset of the formalism that is supported by the current implementation of the net simulator SIMNET. A more complete treatment of the formalism is given in [Ozs85a].

Definition 3.1: An EPTN can be identified as a triple $E=(P,T,A)$ where

$P=\{p_1,p_2,...,p_n\}$ is a set of *places*, $P \neq \phi$

$T=\{t_1,t_2,...,t_n\}$ is a set of *transitions*, $T \neq \phi$

$A \subset \{T \times P\} \cup \{P \times T\}$ is a set of *directed arcs*.

Definition 3.2: A *marking* M of an EPTN, $E=(P,T,A)$, is a mapping

$$M:P \rightarrow N \cup \{0\}$$

where N is the set of natural numbers.

The marking of a place p_i, denoted by $M(p_i)$ represents the number of tokens that reside in place p_i of the net. Tokens in EPTN have the ability to carry data. A token that is carrying data is called a *data token*, else the token is called a *simple token*. The data tokens represent the system temporary entities, for example, jobs, transactions, etc. The simple token is used to indicate the satisfaction of some condition, for example, the availability of a resource.

Definition 3.3: Given an EPTN, $E=(P,T,A)$, the *set of input places* $I(t_i)$ and the *set of output places* $O(t_i)$ for each transition t_i are given by

$I(t_i)=\{p_j \mid (p_j,t_i) \in A\}$

$O(t_i)=\{p_k \mid (t_i,p_k) \in A\}$

Definition 3.4: Each *transition* $t_i \in T$ is defined as a 4-tuple $t_i=(pr, z, q, r)$ where pr is the *precondition* that has to be satisfied for transition t_i to be activated, z is the *transition time* for t_i, q is the *transition procedure* specifying the effects of activating transition t_i, and r is the *output resolution procedure* indicating the procedures to be followed in routing the tokens to alternative places in the set of output places of t_i.

The above definition contains the major extensions to Petri nets. The following sections discusses these extensions and their implications and uses in more detail.

3.3.1. Precondition

In the modeling of systems, more than just conjunctive logic is needed in the specification of the precondition that has to be satisfied before an event could occur. In EPTN, arbitrary logic arrangements on the input places of a transition can be specified using the precondition attribute. For example, assume a transition t_i such that $I(t_i)=\{p_j,p_k\}$. The *AND*, and *OR* logics can be specified on the inputs as follows:

1. $pr(t_i)=(M(p_j)>0) \wedge (M(p_k)>0)$
2. $pr(t_i)=(M(p_j)>0) \vee (M(p_k)>0)$

When the precondition of a transition t_i is true, a set of input places is chosen to participate in the firing of the transition. This set is denoted by $I^{pr}(t_i)$ and is obtained as follows. If $pr(t_i) = (M(p_j)>0 \wedge M(p_k)>0)$ is true, then $I^{pr}(t_i) = \{p_j,p_k\}$. If $pr(t_i) = (M(p_j)>0 \vee M(p_k)>0)$ is true, then there are three possibilities: (1) If $M(p_j)>0$ and $M(p_k)=0$, then $I^{pr}(t_i) = \{p_j\}$, (2) If $M(p_j)=0$ and $M(p_k)>0$ then $I^{pr}(t_i) = \{p_k\}$, and (3) If $M(p_j)>0$ and $M(p_k)>0$, then $I^{pr}(t_i) = \{p_j\}$ or $I^{pr}(t_i) = \{p_k\}$ and the choice is random with equal probability.

3.3.2. Transition Time

Since the formalism is to be used as a performance modeling analysis tool, it is necessary to be able to associate with each event the time it takes. This is achieved in EPTN by including a delay with each transition through the transition time attribute. The transition time may be a constant value for all firings of the transition, or a value from a stochastic function, or a function of a data attribute from the data token that enabled the transition.

3.3.3. Transition Procedure

One of the features of EPTN is the ability of the tokens to carry performance related data. In view of this fact, it is also necessary sometimes for a transition to manipulate and modify these data or perform other operations when it is activated. This is made possible in EPTN by incorporating a transition procedure attribute. This attribute contains instructions on how the data should be manipulated or what other operations should be performed.

3.3.4. Output Resolution

Unlike Petri nets, EPTN allows a subset of the output places to be used in the firing of a transition. Which subset is to be used is usually dependent on some conditions. These conditions, which are expressed in terms of data attributes of data tokens, and their respective actions on the output places are specified in EPTN using the output resolution procedure. For example, if $O(t_i) = \{p_i, p_j, p_k\}$ and if the output resolution procedure $r(t_i)$ is to place a token in p_i when, say, condition A is true and to places p_j and p_k if it is false, could be specified as follows:

$$O^r(t_i) = \begin{cases} \{p_i\} \text{ if condition A is true;} \\ \{p_j, p_k\} \text{ if condition A is false;} \end{cases}$$

In the above specification, $O^r(t_i)$ indicates the output places that are chosen when $r(t_i)$ is evaluated.

With regard to the above notation and terminology, the transition firing rules can now be defined.

Definition 3.4: A *transition* $t_i=(pr, z, q, r)$ of an EPTN is enabled for a marking M if and only of $pr(t_i)$ is true for marking M.

Definition 3.5: The *firing* of a transition $t_i \in T$ of an EPTN, $E=(P,T,A)$, under a marking M in which it is enabled, results in a new marking M' defined as

$$M'(p_j) = \begin{cases} M(p_j)+1 \text{ if } p_j \in O^r(t_i) \wedge p_j \notin I^{pr}(t_i) \\ M(p_j)-1 \text{ if } p_j \in I^{pr}(t_i) \wedge p_j \notin O^r(t_i) \\ M(p_j) \text{ otherwise} \end{cases}$$

3.4. Modeling of Systems Using EPTN

In this section, we make use of a simple but quite realistic example to illustrate the modeling of systems using EPTN. In the discussion, we consider only the modeling aspects and leave out the performance issues. These will be discussed in detail in the next chapter.

In this example, we want to model an on-line information system. The system consists of a number of remote terminals each capable of interrogating a single processor. A customer with a query arrives at one of the terminals and waits in line, if necessary, to use it. When the terminal is available , the customer enters his request, and awaits his reply.

Requests from the terminals are transferred over a network into an input buffer at the processing site where they wait for the CPU. Whenever the CPU is available and there is a request waiting, the CPU removes the first request from the buffer, processes the query, and places the answer into an output buffer where it waits to be transmitted

over the network to the destination terminal. The customer reads the reply and quits his terminal.

Logically, the entire system could be viewed as consisting of three distinct subsystems: the terminal site subsystem, the network subsystem, and the processor site subsystem. In order to model the system using EPTN, the entire system must be viewed as a net which consists of three subnets each one corresponding to a subsystem. Each subnet must in turn be viewed in terms of transitions and places which correspond to the events and conditions that occur in each subsystem, respectively. The places and transitions with their corresponding interpretations for each subnet are described in this section. We assume that there are n terminal sites in the system.

3.4.1. Terminal Site Model

The events that take place at a terminal site s are: the user enters his request (transition tt_1)†, and the user reads the reply from the terminal (tt_2).

The conditions that need to be fulfilled for event tt_1 to occur are: a user has arrived and is waiting for a terminal (place tp_1), and a terminal is available (tp_2). The conditions for event tt_2 are: a user is waiting for a reply (tp_3), and the reply has arrived ($gp_{2,s}$). Note that we have given a different designation to the condition representing the arrival of the reply. This is because the reply will be coming from the processor which will be modeled as a different subnet. So, this place will be shared between two subnets and is therefore designated as a global place.

The conditions that become true as a result of the occurrence of event tt_1 or tt_2 are as follows. When tt_1 occurs, the new conditions are: the user is waiting for a response (which we designated by tp_3), and there is a request waiting to be delivered

† For convenience, tt_1 is used instead of $tt_{1,s}$ unless there is a chance of misunderstanding. The same remark applies to the other site transitions and places.

to the processor ($gp_{1,s}$). Notice that the second place is global. When event tt_2 occurs, that is, the customer reads the reply, the condition that the customer leaves the terminal (tp_4) becomes true.

The subnet for the terminal site model is shown in Figure 3.5. Note that in this model the preconditions and the output resolutions for both events are simple, in the sense that only conjunctive logic is employed. Therefore, we do not specify them explicitly.

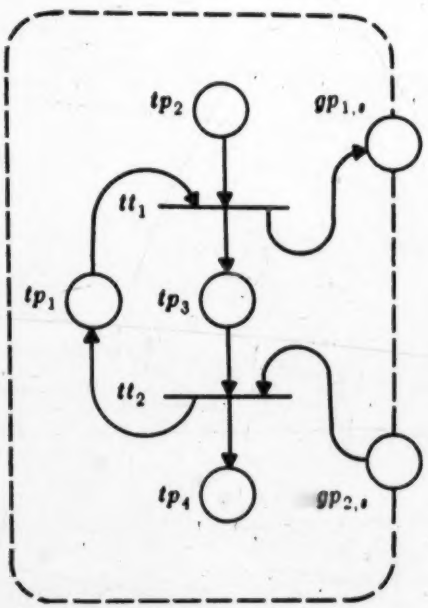

Figure 3.5. Terminal site model.

3.4.2. Processor Site Model

The events that take place at the processor site are the following: a request is received by the CPU from the input buffer (pt_1), a request is processed (pt_2), and a reply is sent to the output buffer (pt_3) to be transmitted to the terminal.

The conditions that need to be satisfied for pt_1 to occur are: the CPU is available (pp_1), and a request has arrived and is waiting in the input buffer (gp_3). The condition for transition pt_2 is that a request has been received by the CPU (pp_2). Finally, the condition for pt_3 is that the request has been processed by the CPU (pp_3).

The output resolution of event pt_1 is pp_2 and that of transition pt_2 is pp_3. The occurrence of event pt_3 results in a reply being placed in the output buffer to be delivered to the terminal (gp_4), and the processor is available again (pp_1). The model is depicted in Figure 3.6.

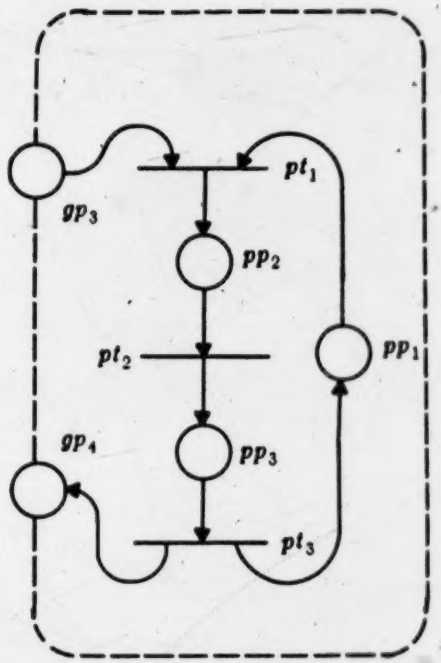

Figure 3.6. Processor site model.

3.4.3. Network Model

Since this is an on-line information system with remote terminals, we need a facility to deliver messages back and forth between the processor and the terminals. This facility is usually provided by a communication network which we are going to model

as a subnet as well.

The events that take place on the network are: a request is transmitted from a terminal to the processor (nt_1), and a reply is transmitted from the processor to a terminal (nt_2). For event nt_1 to occur the network must be available (np_1) and there must be a request waiting to be transmitted at a terminal site ($gp_{1,s}$). The occurrence of the event results in the following conditions becoming true: a request is waiting in the input buffer at the processor (gp_3), and the network is available again(np_1). Similarly, for event nt_2 to occur the network must be available (np_1) and there must be a reply waiting to be transmitted at the processor site (gp_4). The occurrence of the event causes the network to become available again (np_1), and a reply to be delivered to a terminal ($gp_{2,s}$).

Figure 3.7 lists all the places and transitions together with their interpretations for the entire model, and Figure 3.8 shows the entire model with the network connections between the terminal sites and the processor site. Note that when there is more than one terminal, the subnet for the terminal model is replicated. The figure is drawn to clearly show this replication. Also note that the plus sign (+) at transition inputs and outputs signifies a disjunction while the omission of a sign represents conjunction.

Data Token: Customer Information
terminal_ID: The customer terminal number.
time_in: The time the customer entered his request.

Net: On-Line Information System
$gp_{1,s}$: A request from terminal s is waiting to be transmitted over the network to the processor site.
$gp_{2,s}$: A reply from the processor site has been transferred over the network to terminal site s.
gp_3: A request from the network has arrived and is waiting in the input buffer at the processor site.
gp_4: A reply from the processor is waiting in the output buffer at the terminal site to be transmitted over the network to a terminal site.

Subnet : Terminal Site s

tp_1: A terminal is available.
tp_2: A customer has arrived and is waiting for a terminal.
tp_3: A customer is waiting for a reply.
tp_4: A customer quits his terminal.

tt_1: A customer enters his request.
tt_2: A customer reads the reply.

$pr(tt_1) = (M(tp_1)>0) \wedge (M(tp_2)>0)$
$O^r(tt_1) = \{tp_3\}$
$pr(tt_2) = (M(tp_3)>0) \wedge (M(gp_{1,s})>0)$
$O^r(tt_2) = \{tp_2, tp_4\}$

Subnet : Network

np_1: Network is available.

nt_1: A request is transmitted from terminal site to processor site over the network.
nt_2: A reply is transmitted from processor site to terminal site over the network.

$pr(nt_1) = (M(np_1)>0) \wedge (V\{M(gp_{1,z})>0 \mid 1 \le z \le n\})$
$O^r(nt_2) = \{np_1, gp_{2,z}\}$ where $z =$ *terminal_ID* of data token.

Subnet : Processor Site

pp_1: CPU is available.
pp_2: A request has been received by CPU.
pp_3: A request has been processed by CPU.

pt_1: A request is received by CPU from input buffer.
pt_2: A request is processed by CPU.
pt_3: A request is sent to output buffer.

$pr(pt_1) = (M(gp_3)>0) \wedge (M(pp_1)>0)$
$O^r(pt_1) = \{pp_2\}$
$pr(pt_2) = M(pp_2)>0$
$O^r(pt_2) = \{pp_3\}$
$pr(pt_3) = M(pp_3)>0$
$O^r(pt_3) = \{gp_4, pp_1\}$

Figure 3.7. Interpretations for On-Line Information System Model.

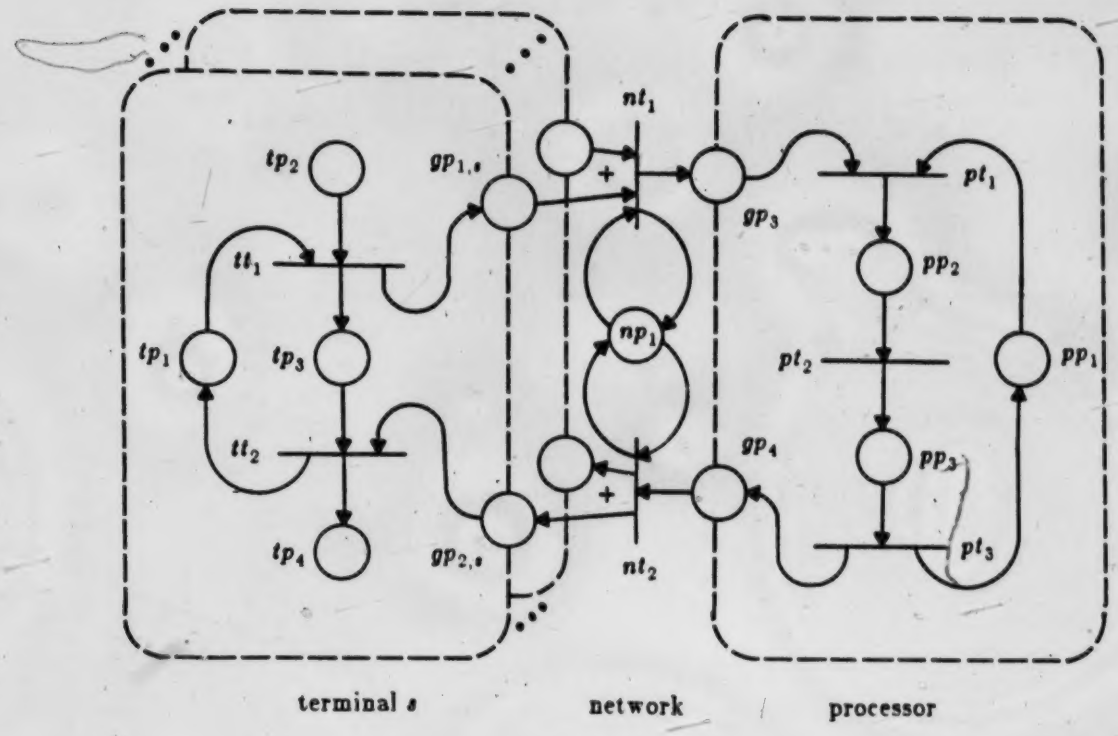

Figure 3.8. EPTN model of On-line Information System.

Chapter 4

SIMNET: An EPTN Model Simulator

SIMNET is a simulation package developed to facilitate the implementation and simulation of systems expressed as extended place/transition nets. SIMNET is implemented as yet another context (see Section 4.2.2) prefixed by the SIMULATION context in SIMULA, and therefore inherits all the features offered by SIMULA as well as those provided by the SIMULATION context. Although all those features are available to the user, only a few of them need to be used in any SIMNET program. In fact, it is possible for someone with no previous experience in SIMULA to write a SIMNET program. SIMNET is very portable and could run on any computer system with a SIMULA compiler. SIMNET is currently running under MTS on an Amdahl 7860 and under UNIX† on a VAX 780‡.

The purpose of this chapter is to provide an overview of SIMNET. For a more formal description of the language, the reader can refer to the User's Guide in Appendix 1. The first section lists the design objectives of SIMNET. After a short overview of the SIMULA language, several features of the SIMNET context are briefly discussed

† UNIX is a trademark of AT&T Bell Laboratories.
‡ VAX is a trademark of Digital Equipment Corporation.

via a complete example.

4.1. Design Objectives

In the design of SIMNET, we had several objectives in mind; most of them have successfully been achieved in the implementation. These objectives are summarized as follows:

1. **User Interface** - The user interface should be friendly, simple and easy-to-use and yet flexible enough to allow any system modeled as EPTN to be fully described.

2. **Random Number Generators** - Different types of number generators should be provided for the simulation of stochastic models.

3. **Accumulating and Reporting of Statistics** - Collecting and reporting of statistics should be automatic. A user should also be provided with the option of writing his own report routine if he/she is not satisfied with either the contents or format of the standard output.

4. **Error Checking and Debugging Facilities** - The simulator should provide error-checking and debugging features to assist the user in the development of the simulation model. A tracing facility should be provided to help trace through the program execution for model verification and for identification of obscure errors.

5. **Modularity and Structuredness** - The user should be able to build models from basic building blocks in a structured fashion.

6. **Control of Simulation** - Commands must be available to allow users to control the internal and external components of the model, to dynamically change the status of the model, and to make several runs sequentially.

4.2. A Short Introduction to SIMULA

This section contains a short introduction to the main features of SIMULA. More detailed description of the language could be found in [DaN66], [Fra77], [Bir81a], [Lam83], and [DMN84]. SIMULA is a general purpose language based on ALGOL. It extends ALGOL by adding two new major language concepts : the *class* and the *context*.

4.2.1. Class

A *class* in SIMULA is used to define concepts that closely resemble those of an abstract data type. Class instances are called objects and several objects of the same class can exist and operate at the same time in the program.

A class declaration is similar to that of a procedure, that is, it has some formal parameters, a local variable declaration and optionally a class body. The general structure for a class declaration is as follows:

```
class class-name(formal-parameters);
   Parameter-specification;
   begin
      Locally declared variables and procedures;
      Class body statements;
   end;
```

The formal parameters and locally declared variables and procedures are called the attributes of the class. An instance of a class could be created by a call to the operator **new** which has the following general structure:

```
new class-name(actual-parameters);
```

When this statement is executed, an instance of the class is created and the class body statements are executed. At the end of the execution, the object is discarded unless there is a pointer referencing it. Referencing is accomplished by first declaring a *reference variable* for the class and then assigning the object instance to it. This is

illustrated below.

```
ref(class-name) reference-variable;
....
reference-variable :- new class-name(actual-parameters);
```

Current data values or procedure definitions of the class attributes could then be accessed from outside the class body by using the *dot notation*, as follows:

```
reference-variable.class-attribute;
```

4.2.2. Prefixed Class

In SIMULA it is possible to build a hierarchical structure of classes by *prefixing*. For example, a class B can be made a subclass of a class A by prefixing class B by A as follows:

```
class A (formal parameters for A);
   Parameter specification;
   begin
      Attributes in A;
      Statements in A;
   end;

A class B (formal parameters for B);
   Parameter specification;
   begin
      Attributes in B;
      Statements in B;
   end;
```

The resulting class obtained is a compound object which is the union of the formal parameters, attributes and statements of the classes in its prefixed sequence together with the structure defined in its own main body. Thus, in the example above, statements in A are executed before statements in B.

4.2.3. Context

A *context* [Bir81b] is a collection of predefined classes, procedures, and data attributes, and statements all contained in a "super class". By prefixing a block with the class name, all these predefined facilities are made accessible within the block body, giving the block a built-in environment in which to operate. SIMULA provides two standard contexts: SIMSET and SIMULATION. SIMSET contains some facilities for list processing. SIMULATION provides some discrete-event simulation features and the process concept. SIMULATION is prefixed by SIMSET, and therefore contains the list processing features of SIMSET as well.

4.3. Programming in SIMNET

SIMNET is implemented as a SIMULA context prefixed by SIMULATION. SIMNET contains, in addition to the discrete-event simulation facilities provided by SIMULATION, predefined classes, procedures and data definitions that could be used : (1) to declare and generate instances of entities, places, transitions, subnets, and nets, (2) to assist the user in finding bugs in a net construction, (3) to simulate a net, (4) to trace the execution of a net, and (5) to automatically collect and report statistics.

In this section, the structure of a SIMNET program is briefly described and the main features are illustrated by means of a programming example.

4.3.1. Program Structure

A SIMNET program is nothing more than a SIMULA program written using the net simulation primitives provided by the SIMNET context. A SIMNET program can be divided into two major parts: a declaration part, and the main program. The declaration part itself consists of six distinct parts: declaration of global constants, declaration of global variables, declaration of other classes and procedures, declaration

of transitions, and declaration of subnets. The main program consists of the net creation statements, and the simulation control statements.

In the rest of this chapter, we will describe these different parts by implementing our On-Line Information System, modeled as an EPTN in the previous chapter, step by step. A complete listing of the program is included in Appendix 2.

In the description, the following conventions will be used: SIMULA keywords will be set in **bold**, SIMNET reserved words will be set in *italics*, user-defined identifiers will be set in UPPER CASE, and comments will be enclosed by '!' and ';'.

4.3.2. Performance Model and Assumptions

First of all, we list the performance model input parameters and assumptions. The input parameters are: mean interarrival time of customers at each terminal site (MIT), mean time to type in a request (MTR), mean time to read a reply (MRR), number of terminal sites in the system (N), message transmission time (T), CPU processing time to receive/transmit a message (P), and mean CPU time to process a request (MTP).

The interarrival time of customers, the time to type in a request, the time to read a reply, and the CPU time to process a request are assumed to be negative exponentially distributed. The number of terminal sites, the message transmission time, and the CPU time to received/transmit a message are assumed to be constant throughout a simulation.

4.3.3. Declaration of Global constants

The declaration of global constants is included after the beginning of the program and before the SIMNET prefixed block. It usually contains the declaration and initialization of input parameters and other global constants used in the simulation. The fol-

lowing SIMNET statements declare the input parameters for our on-line information system and initialize the parameters to values obtained from the standard input file.

```
begin   ! Beginning of main program ;
  ....
  ! Declaration of input parameters ;
  integer N;
  real MIT, MTR, MRR, T, P, MTP;

  ! Initialization of input parameters ;
  Inimage; N := inint;
  Inimage; MIT := inreal;
  Inimage; MTT := inreal;
  Inimage; MTR := inreal;
  Inimage; T := inreal;
  Inimage; P := inreal;
  Inimage; MTP := inreal;;
  ....
  SIMNET begin
    ....
```

4.3.4. Declaration of Global Variables

SIMULA and therefore SIMNET are strongly typed, which means that every global variable used in the program must be declared before it is used. This is done in the global variable declaration part. In our example, we need to declare the global variables GP1, GP2, GP3 and GP4. GP1 and GP2 are two arrays consisting of N places each that link the terminal sites with the network. The places GP3 and GP4 are to global places that link the network with processor site. We also need to declare a reference variable CUST of the class CUSTOMER (see next section) to reference the current customer in the system.

```
ref(place) array GP1(1:N), GP2(1:N);
ref(place) GP3, GP4;
ref(CUSTOMER) CUST;
```

4.3.5. Declaration of Data Objects

As mentioned in Chapter 3, in EPTN tokens are allowed to carry data objects which usually represent temporary entities, for example, transactions, jobs, customers, etc. In a SIMNET program, these data objects are declared in the data object declaration part. As shown below, class CUSTOMER declares a data object that is used to represent a customer in our on-line information system. A CUSTOMER object contains two real value attributes that correspond to two properties of a customer: the time taken to type in a request (TTR) and the time taken to read a reply from the terminal (TRR). Within the object body, TTR and TRR are initialized to random values obtained from negative exponential distributions with means MTR and MRR, respectively. *Seed1* and *seed2* are SIMNET predefined integer variables.

```
dataset class CUSTOMER;
  begin
    real TTR, TRR;

    TTR := Negexp(1/MTR, seed1);
    TRR := Negexp(1/MRR, seed3);
  end;
```

4.3.6. Declaration of Additional Classes and Procedures

This part of a SIMNET program usually contains those additional classes and procedures that are required for the simulation or by the user for some specific purposes. In our example, this part contains the declaration of a class called SOURCE1 and the redefinition of a SIMNET procedure *setcurenv*.

```
source class SOURCE1;
  begin
    ref(CUSTOMER) procedure newdata;
      newdata :- new CUSTOMER;
    real procedure dist;
      dist := Negexp(1/MIT, seed3);
  end;

procedure setcurenv;
```

CUST :- *curdata*;

Class SOURCE1 is declared as a subclass of the SIMNET predefined class *source* to generate customers for the simulation according to a negative exponential distribution. *Source* contains two predefined procedures: *newdata* and *dist*. *Newdata* and *dist* could be redefined to specify the type of data objects that need to be generated and the distribution that is to be used for the generation of the data objects, respectively.

Procedure *setcurenv* is a dummy SIMNET procedure that is invoked by the SIMNET simulator every time a transition is activated. In our example, this procedure is redefined so that the global variable CUST always points to the current data object.

4.3.7. Declaration of Transitions

All transitions that are used to built up a net model are defined in the transition declaration part. A new transition is declared as a subclass of the SIMNET predefined transition class. The transition class contains three predefined procedure attributes which could appropriately be redefined to suit any EPTN transition procedure structures. These procedures are called *precond, work* and *outres* and have the following meanings and default definitions:

1. *Precond* - this boolean function implements the precondition attribute (pr) of an EPTN transition. The default definition is the conjunction (A) of the marking (M) of all the input places to the transition, that is, if $I(t_i) = \{p_i, p_j, p_k\}$ then *precond* = $A(A(M(p_i)>0, M(p_j)>0), M(p_k)>0)$.

2. *Work* - this function procedure is an implementation of two EPTN transition attributes: transition time (z) and transition procedure (q). The body of the WORK procedure implements the EPTN transition procedure while the value returned represents the EPTN transition time. The default definitions are: empty for transition procedure and zero for transition time.

3. *Outres* - this function implements the EPTN output resolution attribute (r). The default definition is the union (U) of all the output places from the transition, that is, if $O(t_i)=\{p_l,p_m,p_n\}$ then *outres* = $U(U(p_l,p_m),p_n)$). U is a SIMNET predefined function that returns a bag of places obtained from the union of its two arguments.

In our on-line information example, we need to declare seven different transitions. We have chosen to illustrate the declaration of only one transition to demonstrate the use of the transition class. The other transitions could be declared in a similar fashion. Notice that preconditions and output resolutions with only conjunctive logics need not be specified although they are specified here for the sake of presentation.

```
transition class TT1(IP1, IP2, OP1, OP2);
  ref(place) array IP1;
  ref(place) IP2, OP1, OP2;
  begin
    integer I;
    ! Precondition attribute ;
    boolean procedure precond;
      begin
        boolean COND;
        COND := FALSE;
        for I := 1 step 1 until N do
          COND := O(COND, M(IP1(I))>0);
        precond := A(COND, M(IP2)>0);
      end;
    ! transition time and procedure attributes ;
    real procedure work;
      work := T;
    ! output resolution attribute ;
    ref(bag) procedure outres;
      outres := U(OP1, OP2);

    ! specification of input places ;
    inpl :- None;
    for I := 1 step 1 until N do
      inpl :- U(inpl, IP1(I));
    inpl :- U(inpl, IP2);
    ! specification of output places ;
    outpl :- U(OP1, OP2);
  end;
```

4.3.8. Declaration of Subnets

The next class of objects that could be defined is *subnet*. A subnet in SIMNET represents a higher-level construct, which could be compared to a subroutine in a program, and is declared in terms of more primitive constructs such as places, transitions and some other facilities such as sources, sinks and resources that are used for performance evaluation purposes. These facilities need to be generated and connected to the appropriate places. In the example shown below, only the declaration of the subnet TERMINAL is illustrated; the other subnets could be declared in a similar way. The actual parameters used in the transition instance generation statements represent the places that link the transitions together.

```
subnet class TERMINAL(IP1, OP1);
  ref(place) IP1, OP1;
  begin
    ref(place) array TP(1:4);
    integer I;

    for I := 1 step 1 until 4 do
      TP(I) :- new place;

    new CER(TP(1), TP(2), TP(3), OP1);
    new CRR(TP(3), IP1, TP(1), TP(4));

    new SOURCE1.connect(TP(2));
    new sink.connect(TP(4));
    new resource(1, "TERM").connect(TP(1));
  end;
```

4.3.9. Net Creation

After the necessary subnets have been declared, instances of the subnets are generated and linked together using global places, if necessary, into the final net. This is done in the net construction part of the main program. In the example below, the net for our on-line information system is constructed from N instances of the subnet TERMINAL, one instance of the subnet NETWORK, and one instance of the subnet PROCESSOR.

```
! Creation of global places ;
for I := 1 step 1 until N do
  begin
    PG1(I) :- new place;
    PG2(I) :- new place;
  end;
  PG3 :- new place;
  PG4 :- new place;

! creation of subnets ;
for I := 1 step 1 until N do
  new TERMINAL(PG2(I), PG1(I));
new NETWORK(PG1, PG4, PG3, PG2);
new PROCESSOR(PG3, PG4);
```

4.3.10. Simulation Control Statements

Now that the net is constructed, it is time to check the net for bugs, start the simulation, validate the model by tracing the execution, and report the statistics. These could easily be done by invoking the predefined SIMNET procedures: *check_net, display_net, simulate, trace,* and *report.* Invocation of the procedure *check_net* causes the net to be checked for any unconnected places and/or transitions and similar syntactic errors. The procedure *display_net* causes the structure of the net to be listed. The simulation can be started by the command *simulate* . Parts of the simulation can be traced by turning the trace feature on using the *trace* command. The output is reported using the *report* command. The use of these procedures are illustrated in the example below.

```
check_net;
display_net;
trace(1, 25, 'J');
simulate(60, 'T');
report;
```

A sample of the output produced is as follows:

```
--- NET IS APPARENTLY OKAY ---

NET STRUCTURE
-------------
PLN = LOCAL PLACE N; PGN = GLOBAL PLACE N
TT1 IN TERMINAL 1 : IN={PL1,PL2}, OUT={PL3,PG1}
TT2 IN TERMINAL 1 : IN={PL3,PG2}, OUT={PL1,PL4}
TT1 IN TERMINAL 2 : IN={PL1,PL2}, OUT={PL3,PG3}
TT2 IN TERMINAL 2 : IN={PL3,PG4}, OUT={PL1,PL4}
TT1 IN TERMINAL 3 : IN={PL1,PL2}, OUT={PL3,PG5}
TT2 IN TERMINAL 3 : IN={PL3,PG6}, OUT={PL1,PL4}
NT1 IN NETWORK 4 : IN={PG1,PG3,PG5,PL1}, OUT={PG7,PL1}
NT2 IN NETWORK 4 : IN={PG8,PL1}, OUT={PG2,PG4,PG6,PL1}
PT1 IN PROCESSOR 5 : IN={PG7,PL1}, OUT={PL2}
PT2 IN PROCESSOR 5 : IN={PL2}, OUT={PL3}
PT3 IN PROCESSOR 5 : IN={PL3}, OUT={PG8,PL1}

TRACE BEGINS . . .

   TIME          EVENT(S)

   2.8314   TERMINAL 2 - CUSTOMER 1 ARRIVES
            TERMINAL 2 - TT1 IS ENABLED BY CUSTOMER 1
   3.6609   TERMINAL 1 - CUSTOMER 2 ARRIVES
            TERMINAL 1 - TT1 IS ENABLED BY CUSTOMER 2
   4.6665   TERMINAL 3 - CUSTOMER 3 ARRIVES
            TERMINAL 3 - TT1 IS ENABLED BY CUSTOMER 3
   5.7664   TERMINAL 2 - TT1 FIRES
            NETWORK 4 - NT1 IS ENABLED BY CUSTOMER 1
            NETWORK 4 - NT1 FIRES
            PROCESSOR 5 - PT1 IS ENABLED BY CUSTOMER 1
            PROCESSOR 5 - PT1 FIRES
            PROCESSOR 5 - PT2 IS ENABLED BY CUSTOMER 1
            PROCESSOR 5 - PT2 FIRES
            PROCESSOR 5 - PT3 IS ENABLED BY CUSTOMER 1
            PROCESSOR 5 - PT3 FIRES
            NETWORK 4 - NT2 IS ENABLED BY CUSTOMER 1
   5.7671   NETWORK 4 - NT2 FIRES
            TERMINAL 2 - TT2 IS ENABLED BY CUSTOMER 1
   6.2181   TERMINAL 2 - TT2 FIRES
            TERMINAL 2 - CUSTOMER 1 TERMINATES
   7.1589   TERMINAL 3 - CUSTOMER 4 ARRIVES
   8.8589   TERMINAL 1 - CUSTOMER 5 ARRIVES

TRACE ENDS.
```

SIMULATION TIME : 60.00

RESOURCE	USAGE	TPUT	QLENGTH	QTIME
TERM - TERMINAL 1	76.79582	0.23333	1.78333	7.64286
TERM - TERMINAL 2	79.92789	0.26667	1.96667	7.37500
TERM - TERMINAL 3	71.45123	0.16667	0.71667	4.30000
NET - NETWORK 4	0.00333	0.66667	0.00000	0.00000
CPU - PROCESSOR 5	0.01085	1.00000	0.00000	0.00000

NUMBER OF CUSTOMER RECEIVED = 31
NUMBER OF CUSTOMER COMPLETED = 20
NUMBER OF CUSTOMER IN QUEUES = 11
MEAN RESPONSE TIME = 11.76761
VARIANCE = 89.90717

Chapter 5

The Algorithms

The two locking-based concurrency control algorithms studied in this thesis are presented in this chapter. The first one is the *Centralized Locking Algorithm with Hole lists* of Garcia-Molina [Gar79] and the second one is a modified version of the *Distributed Locking Algorithm* due to Gardarin and Chu [GaC80].

5.1. Assumptions

Before describing the algorithms, we list the assumptions under which the two algorithms were designed to operate reliably. Some of the assumptions have already been mentioned in the previous chapters, but we list them here for completeness.

1. The database is static, that is, no data items are added to or deleted from the database.
2. Messages are not lost or duplicated in the network.
3. Intersite message transmission is error-free.
4. Network partitioning never occurs.

5. Site failures are "clean", that is, when a site fails it halts all activities; when it recovers, it always detects the failure and initiates the recovery routines.

6. Site failures are detected by the communication subsystem within a reasonable time period.

7. Transactions initially specify the items that they will reference. This permits the deadlock avoidance technique to be used for deadlock resolution.

8. Undo/redo logs and write-ahead protocols are used to prepare sites for recovery.

9. The database is fully replicated at all sites.

10. A point-to-point type of network is used.

In the following algorithms, the site where an update transaction originates is called the *originating site* and is also the *coordinator* for the transactions that originate at the site; the other sites that cooperate in the execution of the transaction are referred to as the *cohort sites* or *participating sites*.

In the centralized locking algorithm, the site which is responsible for the allocation and deallocation of locks to transactions is called the *central site*. The *done-set* refers to the set of all the updates that have been completed at that site. The *hole-set* is the set of all transactions that are active, that is, have obtained their locks but have not yet released all of them.

The transaction *base-set* is the set of all items that are referenced by the transaction. The transaction *write-set* is the set of data items that are modified by the transaction.

We now describe the two algorithms. Several of the techniques discussed in the Chapter 2 are being applied. Since we are mainly interested in the performance of the algorithms rather than in the analysis of their correctness, some of the details are omitted.

5.2. The Resilient Centralized Locking Algorithm

The main techniques that are used in this algorithm are:

1. Two-phase locking is used for concurrency control.
2. Lock management is centralized.
3. A two-phase commit protocol is used to achieve resiliency against failures.
4. An election protocol is used to elect a new central site if the central site fails.

A brief description of the algorithm follows and a more formal description is given in Appendix 3.

Algorithm CL

Step 1. When a transaction arrives at a site s, the site requests locks for all the items in the transaction base-set from the central site.

Step 2. The central site checks all the requested locks. If an item lock has already been granted, the request is queued in the item queue; otherwise the item is locked by the request. In order to avoid deadlock, requests are made in some predefined order on the items and requests can wait for only one item at a time.

Step 3. When all the requested locks have been granted, the central site assigns a sequence number to the transaction, appends it to a copy of the hole-set and sends everything together with a *locks-granted* message to the originating site s.

Step 4. When the originating site receives the *locks-granted* message, it checks the hole-set and the site done-set to see if there is an older conflicting transaction (that is, with smaller sequence number that is not a member of the union of the hole-set and the done-set). If so, the transaction is delayed until all older

conflicting transactions are completed; otherwise the base-set for the transaction is read and the new update values computed. site s then sends a *ready-to-commit* message together with the new update values to the cohort sites.

Step 5. When a cohort site receives a *ready-to-commit* message, it checks if the transaction can be committed. If so, it saves the transaction's new update values in stable storage and acknowledges receipt of the message to site s; otherwise, it sends an *abort* message to site s.

Step 6. If all up sites have agreed to commit, site s sends a *commit* message to all cohort sites; otherwise it sends an *abort* message to all sites and releases all the locks held by the transaction.

Step 7. If an *abort* message is received by a cohort site, the site aborts the transaction.

Step 8. If a cohort site receives a *commit* message, it performs the update on its local database. The central site also releases the locks held by the transaction.

Step 9. The transaction terminates.

5.2.1. Site Model for CL algorithm

The EPTN model for a site s is shown in Figure 5.1. In the model, we do not distinguish between a central site and a non-central site. Every site in the system can be a central site. However, it is only when a site is acting as a central site that the central site events (that is, events that take place at the central site only) could occur. The events that take place at each site are described below. In the description, central site events are indicated with a superscript c (^{c}).

The events are : request locks (st_1), check conflict (st_2^c), put in item queue (st_3^c), remove from item queue (st_4^c), lock data items (st_5^c), send lock granted message to originating site (st_6^c), check if hole set contains any older conflicting transaction (st_7), put

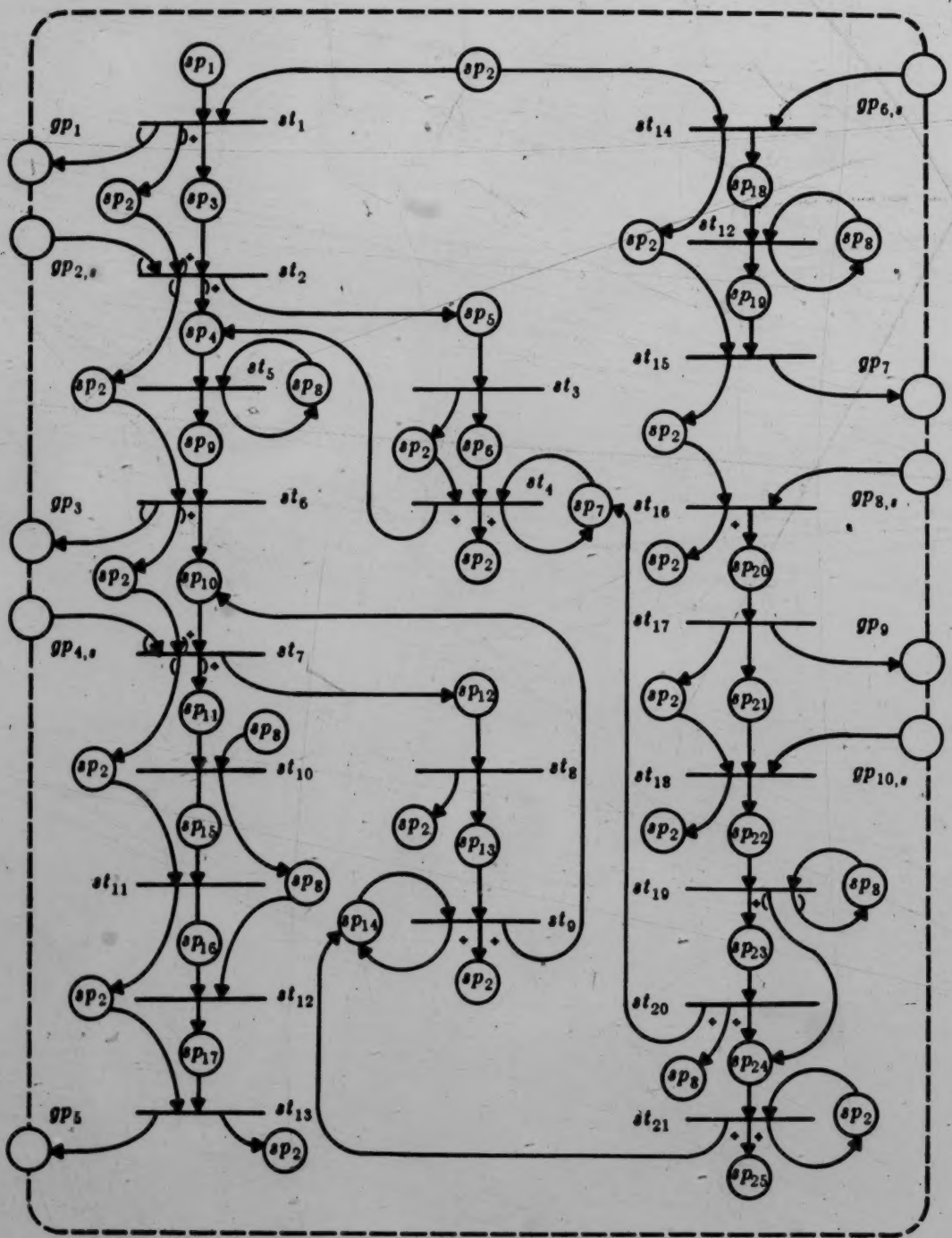

Figure 5.1. Site model for CL algorithm.

in wait queue (st_8), remove from wait queue (st_9), read read-set (st_{10}), compute new update values (st_{11}), log update values (st_{12}), broadcast intend to update message to all sites (st_{13}), receive intend to update message (st_{14}), send acknowledgment (st_{15}), receive acknowledgment (st_{16}), broadcast commit message (st_{17}), receive commit message (st_{18}), update database (st_{19}), unlock data items (st_{20}), and terminate transaction (st_{21}).

The conditions are as follows: a local transaction has arrived (sp_1), CPU is available (sp_2), lock request is waiting to be sent over the network to the central site (gp_1), a local lock request is waiting (sp_3), a global foreign lock request has arrived ($gp_{2,s}$), conflict does not exist (sp_4), conflict exists (sp_5), transaction is waiting for locked items (sp_6), data items are unlocked (sp_7), I/O is available (sp_8), data items are locked (sp_9), lock granted message is waiting to be sent to originating site (gp_3), local transaction has received all required locks (sp_{10}), lock granted message has arrived from central site ($gp_{4,s}$), hole set contains no older conflicting transactions (sp_{11}), hole set contain older conflicting transactions (sp_{12}), transaction is waiting for older conflicting transaction to complete (sp_{13}), older conflicting transaction has completed (sp_{14}), read-set is read (sp_{15}), new update values are computed (sp_{16}), update values are stored in log (sp_{17}), intend to update message is waiting to be transmitted to all sites (gp_5), a lock granted message has arrived ($gp_{6,s}$), lock granted message is received (sp_{18}), update values are stored in log (sp_{19}), acknowledgment is waiting to be transmitted to originating site (gp_7), an acknowledgment has arrived ($gp_{8,s}$), an acknowledgment is received (sp_{20}), transaction is ready to be committed locally (sp_{21}), a commit message is waiting to be transmitted to all sites (gp_9), a commit message has arrived ($gp_{10,s}$), transaction is committed (sp_{22}), database is updated (sp_{23}), data items are unlocked (sp_{24}), and transaction has terminated (sp_{25}).

Given $T=\{st_1,...,st_{21}\}$ and $P=\{sp_1,...,sp_{25},gp_1,\ldots,gp_{10}\}$ then the following are true about the preconditions of the transitions :

1. $pr(st_2) = ((M(gp_{2,s})>0 \wedge M(sp_2)>0) \vee M(sp_3)>0)$
2. $pr(st_7) = ((M(gp_{4,s})>0 \wedge M(sp_2)>0) \vee M(sp_{10})>0)$
3. $\forall st_i \in T(i \notin \{2,7\} \Rightarrow pr(st_i) = \wedge \{M(p_j)>0 \mid p_j \in I(st_i)\})$

Given T and P as defined above, the following are true about the output resolution of the transitions:

1. $O^r(st_1) = \begin{cases} \{gp_1,sp_2\} & \text{if transaction is local} \\ \{sp_3\} & \text{otherwise} \end{cases}$
2. $O^r(st_2) = \begin{cases} \{sp_2,sp_4\} & \text{if conflict exists} \\ \{sp_5\} & \text{otherwise} \end{cases}$
3. $O^r(st_4) = \begin{cases} \{sp_4,sp_2,sp_7\} & \text{if more transactions are waiting for data items} \\ \{sp_4,sp_2\} & \text{otherwise} \end{cases}$
4. $O^r(st_6) = \begin{cases} \{gp_3,sp_2\} & \text{if transaction is local} \\ \{sp_{10}\} & \text{otherwise} \end{cases}$
5. $O^r(st_7) = \begin{cases} \{sp_{12}\} & \text{if hole set contains no older conflicting transactions} \\ \{sp_2,sp_{11}\} & \text{otherwise} \end{cases}$
6. $O^r(st_9) = \begin{cases} \{sp_{10},sp_{14}\} & \text{if more conflicting transactions are waiting} \\ \{sp_2,sp_{10}\} & \text{otherwise} \end{cases}$
7. $O^r(st_{19}) = \begin{cases} \{sp_{23}\} & \text{if site is central} \\ \{sp_5,sp_{24}\} & \text{otherwise} \end{cases}$
8. $O^r(st_{20}) = \begin{cases} \{sp_5,sp_7,sp_{24}\} & \text{if transactions are waiting for data items} \\ \{sp_5,sp_{24}\} & \text{otherwise} \end{cases}$
9. $O^r(st_{21}) = \begin{cases} \{sp_{14},sp_{23}\} & \text{if younger conflicting transactions are waiting} \\ \{sp_2,sp_{23}\} & \text{otherwise} \end{cases}$

5.2.2. Network Model for CL Algorithm

The network model that connects the sites together is shown in Figure 5.2.

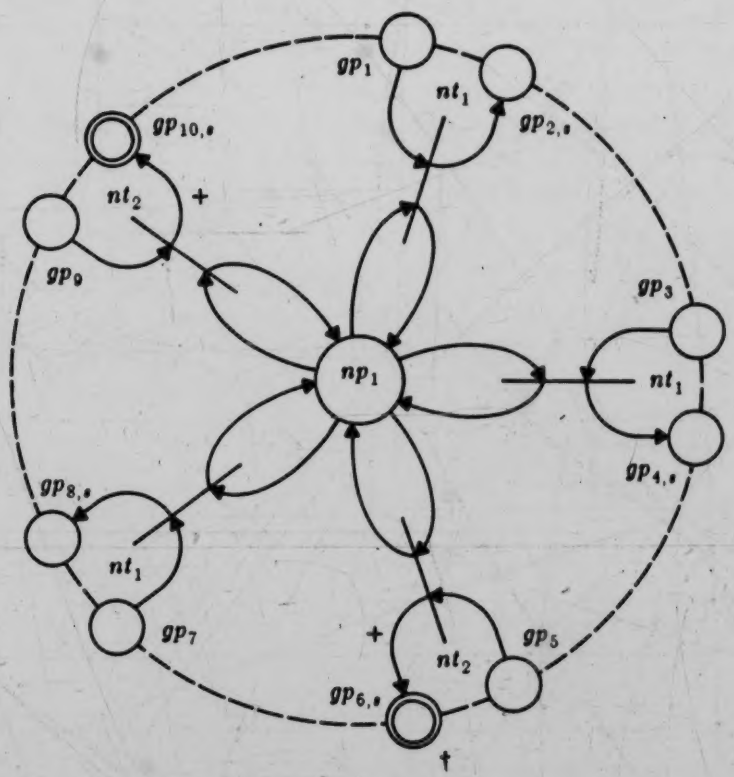

Figure 5.2. Network model for CL algorithm.

(† The symbol (double circle) represents a replication of n places.
The plus sign (+) on the link indicates disjunctive logic on the n places.)

Only two types of events can occur on a network: send a message from a site to another site (nt_1), and send a message from a site to all the other sites (nt_2).

The conditions are: the network is available (np_1), a message is waiting to be transferred to the destination site (gp_1,gp_3,gp_5,gp_7,gp_9)‡, and the message has been transferred ($gp_{2,*}$,$gp_{4,*}$,$gp_{6,*}$,$gp_{8,*}$).

‡ These places are global and have the same interpretations as those described for the site model.

Given $T=\{nt_1,nt_2\}$ and $P=\{np_1,gp_1,gp_{10,s}\}$, then the following are true about the preconditions and output resolutions of the transitions.

1. $pr(nt_1) = (M(np_1)>0) \wedge (V\{M(p_j)>0 \mid p_j \in I(nt_1)\})$
2. $pr(nt_2) = (M(np_1)>0) \wedge (V\{M(p_j)>0 \mid p_j \in I(nt_2)\})$
3. $O^r(nt_1) = \{p_j\}$ where p_j is the destination place
4. $O^r(nt_2) = O(nt_2)$

5.2.3. Failure Considerations

Failure of a Coordinator Site: When the central site discovers that the coordinator for a transaction t has failed, it starts a termination protocol which allows the transaction to be terminated and its locks freed. The termination protocol is fully described in [Gar79] and consists of three phases: one phase to inquire about the transaction at all sites and two phases to actually cancel or force execution of the transaction at all sites. Below we give a very brief outline of the protocol.

In the first phase, the central site sends a *have-you-seen-t* message to all sites. When a site s receives the message, it checks if it has received a *commit* message for transaction t or has actually performed t and replies to the central site with either a *have-seen* or a *have-not-seen* message. The site also makes a commitment not to acknowledge any *ready-to-commit* message for t it might receive later.

If the central site receives at least one *have-seen* message, it will force the performance of transaction t at all sites in the system. To achieve this, it uses a two-phase commit protocol. First, the central site sends a *force-execution* message and the new update values for t, obtained from the sites that have seen the transaction, to all sites. When a site receives the message, it saves the update values in stable memory and sends an acknowledgment to the central site. After the central site has received acknowledgments from all the up sites, it sends a *commit* message to all the sites. It also

performs the update locally and releases the locks held by the transaction. All the other sites perform the update on their local database when they receive the *commit* message.

If the central site receives *have-not-seen* messages from all the up sites and only the coordinator has failed, it will cancel the transaction at all sites. Cancellation of a transaction is also performed in two phases. In the first phase, the central site sends a *ready-to-cancel* message for transaction t to all sites. When a site receives such a message, it records in its log that transaction t is to be canceled and acknowledges to the central site. After every site has acknowledged, the central site sends a *cancel* message to all sites and releases the locks. The other sites also cancel the transaction locally after receiving the *cancel* message or after the timeout period expires.

Failure of a Cohort site: Failure of a cohort site does not pose any problem to the correct operation of the algorithm. The operational sites will continue to operate normally.

Failure of the Central Site: When the central site fails, the entire system crashes. Every site goes into a failure mode and stops receiving transactions. The operational sites then group together and start an election protocol to elect a new central site. Election protocols are discussed in [Gar79] and [Gar82]. A brief description of the protocol is as follows.

When a site x discovers that the central site has failed, it goes into failure mode and stops receiving transactions. Then after every t_1 seconds it sends an *are-you-up* message to all sites and constructs an "active table" from the acknowledgments received (the other sites might be doing the same process simultaneously). If the site thinks that it is a candidate for the central site, that is, it has the highest site number among the functional sites, it sends a *propose-to-become-central-site* message to all sites. When a site receives such a message, it sends a *vote* to site x if it thinks that x

can become a central site. It also makes a commitment not to vote for any other sites.

If after t_2 seconds, site x has been able to collect a majority of votes, it sends an *I-am-the-new-central-site* to all sites and waits for acknowledgments; otherwise, it sends an *I-did-not-make-it* message to all sites and starts the election protocol all over again. When a site receives an *I-am-the-new-central-site* message it stores the number of the new central site in its configuration table and acknowledges receipt of the message. If an *I-did-not-make-it* message is received or a site fails during the election, the election protocol is restarted.

After the election, the new central site terminates all unfinished transactions before it resumes normal operation.

5.2.4. Recovery Considerations

Recovery of a non-central site: When a non-central site s recovers from a failure, it inspects its log and undoes/redoes any unfinished transactions. Then it sends a *I-would-like-to-recover* message to all sites. If the sites are in the middle of an election, the recovering site will be told to defer its recovery until the election is over. Otherwise, the central site sends to the site the log of all updates that it missed while it was down. All the other active sites also start sending messages to site s.

Recovery of a central site: When a central site recovers from a failure, it stops acting as a central site and follows exactly the same recovery procedures as for a non-central site.

5.3. The Resilient Distributed Locking Algorithm

The main techniques that are used in this algorithm are:

1. Management of locks is done at all sites. In order to achieve synchronization among sites, global timestamps are used (global timestamps are constructed by

concatenating the local time with the site number).

2. Two phase-locking is used for concurrency control.

3. A two-phase commit protocol is used to achieve resiliency against site failures.

A brief description of the algorithm follows. A formal description is included in Appendix 4.

Algorithm DL

Step 1. When a transaction t arrives at a site s, it is assigned a unique global timestamp.

Step 2. Site s then requests locks for all the items in the transaction write-set from all the other active sites. At the same time it attempts to lock the items in the transaction base-set locally.

Step 3. When a site attempts to grant locks to transaction t, it checks all the requested locks. If an item is already locked, it checks the timestamp and state of the transaction holding the lock. If it is younger than transaction t, and it is not in the *ready-to-commit* or *commit* state, it is canceled and transaction t obtains the lock for the item. Otherwise, t is inserted in the item wait queue in ascending order of timestamp value to avoid deadlock.

Step 4. When all the locks requested by a transaction could be granted, the site sends a *locks-granted* message to the originating site s.

Step 5. When site s has obtained *locks-granted* messages from all the up sites, it reads the transaction base-set and computes the new update values. Site s then sends a *ready-to-commit* message together with the newly computed update values to all the cohort sites.

Step 6. When a cohort receives a *ready-to-commit* message, it saves the update values in stable memory and acknowledges receipt of the message to site s.

Step 7. When site s has received acknowledgments from all the up sites s, it decides on whether to commit or abort the transaction and sends its decision to the cohort sites.

Step 8. If a cohort site receives a *commit* message, it performs the update on its local database and releases the locks held by the transaction.

Step 9. If an *abort* message is received the transaction is aborted and its locks released.

Step 10. The transaction terminates.

5.3.1. Site Model for DL algorithm

The site model for the DL algorithm is shown in Figure 5.3. The events that take place at each site are as follows: request locks (st_1), check conflict (st_2), put in item queue (st_3), remove from item queue (st_4), lock data items (st_5), send lock granted message (st_6), receive lock granted messages (st_7), read read-set (st_8), compute new update values (st_9), log update values (st_{10}), broadcast intend to update message to all sites (st_{11}), receive abort message (st_{12}), cancel transaction (st_{13}), receive intend to update message (st_{14}), send acknowledgment (st_{15}), receive acknowledgement (st_{16}), broadcast commit message (st_{17}), receive commit message (st_{18}), update database (st_{19}), unlock data items (st_{20}), and terminate transactions (st_{21}).

The conditions are as follows: a local transaction has arrived (sp_1), CPU is available (sp_2), lock request is waiting to be sent over the network to all sites (gp_1), a local lock request is waiting (sp_3), a global foreign lock request has arrived ($gp_{2,s}$), abort message is waiting to be sent to all sites (gp_3), transaction is to be aborted locally (sp_4), conflict does not exist (sp_4), conflict exists (sp_6), transaction is waiting for locked

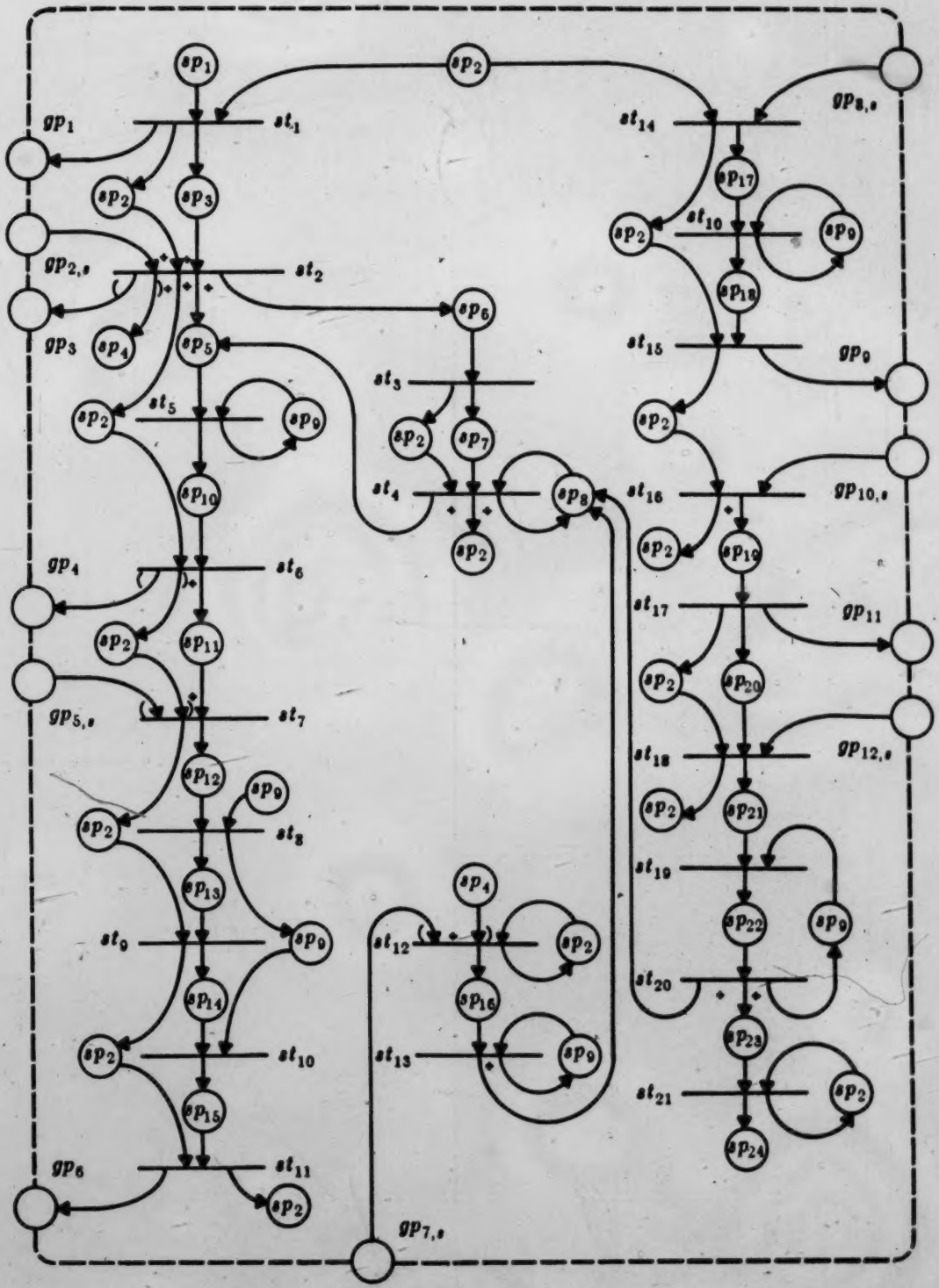

Figure 5.3. DL algorithm site model.

items (sp_7), data items are released (sp_8), I/O is available (sp_9), data items are locked (sp_{10}), lock granted message is waiting to be sent to originating site (gp_4), local transaction has acquired all required locks (sp_{11}), lock granted message has arrived from a site ($gp_{5,s}$), lock granted messages have been received from all up sites (sp_{12}), read-set is read (sp_{13}), new update values are computed (sp_{14}), update values are stored in log (sp_{15}), intend to update message is waiting to be transmitted to all sites (gp_6), an abort message has arrived (gp_7), transaction is to be canceled (sp_{16}), a lock granted message has arrived ($gp_{8,s}$), lock granted message is received (sp_{17}), update values are stored in log (sp_{18}), acknowledgment is waiting to be transmitted to originating site (gp_9), an acknowledgment has arrived ($gp_{10,s}$), an acknowledgment is received (sp_{19}), transaction is ready to be committed locally (sp_{20}), a commit message is waiting to be transmitted to all sites (gp_{11}), a commit message has arrived ($gp_{12,s}$), transaction is committed (sp_{21}), database is updated (sp_{22}), data items are unlocked (sp_{23}), and transaction has terminated (sp_{24}).

Given $T=\{st_1,...,st_{21}\}$ and $P=\{sp_1,...,sp_{24},gp_1,\ldots,gp_{12,s}\}$ then the following are true about the preconditions of the transitions :

1. $pr(st_1) = ((M(gp_{2,s})>0 \ V \ M(sp_3)>0) \wedge M(sp_2)>0))$
2. $pr(st_7) = (M(gp_{5,s})>0 \wedge M(sp_2)>0) \ V \ M(sp_{11})>0))$
3. $pr(st_{12}) = ((M(gp_{7,s})>0 \ V \ M(sp_4)>0) \wedge M(sp_2)>0))$
4. $\forall st_i \in T(i \notin \{2,7\} \Rightarrow pr(st_i) = \wedge\{M(p_j)>0) \mid p_j \in I(st_i)\})$

Given T and P as defined above, the following are true about the output resolution of the transitions:

1. $O^r(st_2) = \begin{cases} \{gp_3,sp_4,sp_2\} & \text{if abort} \\ \{sp_6\} & \text{if conflict} \\ \{sp_2,sp_5\} & \text{otherwise} \end{cases}$

2. $O^r(st_4) = \begin{cases} \{sp_5,sp_8\} & \text{if more transactions are waiting for data items} \\ \{sp_5,sp_2\} & \text{otherwise} \end{cases}$

3. $O^r(st_6) = \begin{cases} \{sp_{11}\} & \text{if transaction is local} \\ \{gp_4, sp_2\} & \text{otherwise} \end{cases}$

4. $O^r(st_7) = \begin{cases} \{sp_2\} & \text{if lock granted messages have arrived from all up sites} \\ \{sp_2\} & \text{otherwise} \end{cases}$

5. $O^r(st_{13}) = \begin{cases} \{sp_8, sp_9\} & \text{if more transactions are waiting for data items} \\ \{sp_8\} & \text{otherwise} \end{cases}$

6. $O^r(st_{21}) = \begin{cases} \{sp_8, sp_{23}, sp_9\} & \text{if transactions are waiting for data items} \\ \{sp_{23}, sp_9\} & \text{otherwise} \end{cases}$

5.3.2. Network Model for DL algorithm

The network model for the DL algorithm is shown in Figure 5.4.

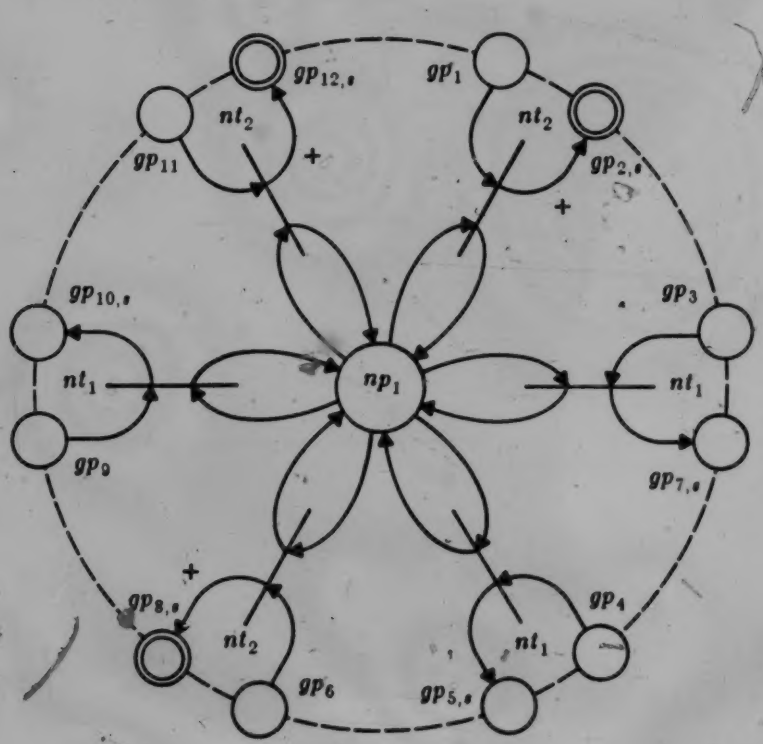

Figure 5.4. Network model for DL algorithm.

The events, conditions, and preconditions and output resolutions for the transitions are similar to those described in Section 5.2.2.

5.3.3. Failure Considerations

Failure of a coordinator: When a coordinator fails, the cohort sites elect a new coordinator to terminate unfinished transactions. The same election and termination protocols used in the centralized locking algorithm (see Section 3.1.1) could be used to elect a new coordinator and to terminate unfinished transactions respectively.

Failure of a cohort site: Failure of a cohort site does not pose any problem to the correct operation of this algorithm.

5.3.4. Recovery Considerations

When a site recovers from a failure, it first reconstructs a consistent database by undoing/redoing unfinished transactions recorded in its log. It then chooses an operative site which has an up-to-date version of the replicated data and sends an *I-would-like-to-recover* message to that site. When the site receives such a message, it sends the log of missing updates, the lock tables and the message queues to the recovering site. It also informs all the other sites that site s is up. The other sites will resume communication with site s and start sending messages to it.

Chapter 6

Simulation Model

In this chapter, we present the simulation model, the input parameters for the model, and the performance metrics that are measured.

6.1. Simulation Models

In order to obtain performance results for the two algorithms, simulation model programs for the algorithms are developed and then executed on a computer. In this study, two different simulation approaches are used: discrete event simulation (DES) ([Fra77] and [Gor78]) and EPTN simulation.

In the DES approach, significant events in the algorithms are first identified. The action that takes place in each event is then implemented as a process in SIMULA. Typical events are: arrival of a job, sending a lock request to another site, setting locks on some data items, etc. In SIMULA, a list of events ordered by the time of occurrence is maintained. Whenever a process is scheduled, the process is entered on the event list to be activated at the appropriate time. When the simulation is run, the program cycles through the list of events and performs the folbwing operations: (1) select the event with the earliest time, (2) set the simulation clock to this time, and (3) perform

the action. Sometimes the action performed by one event could cause another event to be scheduled. The simulation continues as long as there are events to be executed. To drive the simulation a job generator is used. The generator creates and schedules new events which represent the arrival of new jobs in the system.

In the EPTN simulation approach, the tasks are much simpler. All that need to be done are: (1) using SIMNET, implement the EPTN models for the algorithms by declaring the subnets for the sites and networks and then linking them together as described in Chapter 4; (2) declare and generate the sources, sinks, and resources, that are needed for performance evaluation purposes, and connect them to the appropriate places; and (3) run the simulation to obtain performance results.

In this study, the following assumptions about the simulation models are made. Every site in the system has only one CPU server and one I/O server. The servers are implemented at a very high level and are visible only through the processing time they take to process a request. Each server has an associated queue and executes requests one at a time using a FCFS (First-Come-First-Served) discipline. Requests for service can arrive at a site from three sources: the users at the site, the network, and the site itself. Requests that need both CPU and I/O services are processed in the order: I/O service followed by CPU service with no interleaving between the two. For the network model, we assume that there is a communication link between every two sites and that the bandwidth is large enough to handle several messages at one time.

6.2. Input Parameters

The model input parameters are:

1. *Mean interarrival time of transactions* (I_t) : The arrival time of transactions to each site is considered to be negative exponentially distributed with a mean interarrival time I_t.

2. *Mean base-set size* (B_s) : The base-set represents the set of items that are referenced by a transaction. The base-set size is assumed to be negative exponentially distributed with a mean base-set size of B_s. The write-set size refers to the set of items that are modified by a transaction. The write-set size is assumed to be uniformly distributed between 1 and the base-set size.

3. *Number of sites in the system* (N) : The number of sites in the system is assumed to be N.

4. *Message processing time* (P) : The message processing time is the CPU time it takes to transmit or received a message over the network.

5. *Message transmission time* (T) : The message transmission time represents the delay that is incurred in transmitting a message from a site to another site. The transmission time is assumed to be constant regardless of the message length of network load.

6. *CPU time slice* (C_s) : The CPU time slice is the time it takes a CPU server to do some computations, for example, to modify or check a lock or to compare two values.

7. *CPU compute time* (C_c) : This is the amount of time that is needed per base-set item to compute a new update value. For example, if B is the base-set size of a transaction, the total time required to compute all the update values for the transaction is $B \times C_c$.

8. *I/O time slice* (IO_s) : The I/O time slice is the time needed to read or write a lock or a timestamp from/to the secondary storage.

9. *I/O update time* (IO_e) : This is the time it takes to read or write an item from of to the secondary storage.

10. *Database size* (D) : The database size is the total number of data items in the database.

11. *Mean time between failures* ($MTBF$) : The time between failures for a site is assumed to be negative exponentially distributed with a mean of $MTBF$.

12. *Mean time to repair* ($MTTR$) : The time it takes a site to recover from a failure is assumed to be negative exponentially distributed with a mean of $MTTR$.

The default value and range for each parameter are listed in the table below. A series of dots "..." is used to indicate that the parameter value is not varied.

Table 6.1. Input parameter values.

Parameter	Default value	Range
I_t	10 seconds	1-20 seconds
B_t	2 data items	2-15 data items
N	6 sites	2-15 sites
P	0.003 second	...
T	0.1 second	0.005-0.25 second
C_c	0.001 second	...
C_s	0.00001 second	...
IO_e	0.025 second	0.0-0.05 second
D	1000 data items	25-1500 items
$MTBF$	37 hours	5-37 hours
$MTTR$	3480 seconds	...

6.3. Performance Metrics

The performance metrics that are observed during each simulation run include the following:

1. *Mean response time* : The response time of a transaction is defined as the interval of time between the arrival of the transaction at a site and its completion at the same site. A transaction is assumed to be completed when the originating site has successfully done all the work requested by the transaction. The mean response time is the average of the response times for all successfully completed transactions.

2. *I/O and CPU utilizations* : The utilization of a server at a site is defined as the percentage of the available time that the server is busy.

3. *Availability* : In this study, the availability is defined as the percentage of the total time that the system is available for the processing of update transactions.

4. *Mean number of Messages* : This represents the average number of messages that are exchanged among sites per update transaction.

6.4. Conflict Resolution

During a simulation, a conflict between a new transaction T_i and a currently active transaction in the system occurs if the following relational expression[Ozs85a] is true:

$$r_u < \frac{\left(\begin{matrix} D - AT \\ B_i \end{matrix}\right)}{\left(\begin{matrix} D \\ B_i \end{matrix}\right)}$$

Where r_u is a uniformly distributed random variable between 0 and 1,

B_i is the base-set size of transaction T_i,

AT is the total number of data items locked by active transactions in the system,

D is the database size, and

$\binom{x}{y}$ represents the number of ways y items can be chosen from x items.

Whenever a conflict occurs, one of the currently active transactions is randomly chosen as the blocking transaction. The action taken after a conflict depends on the algorithm being simulated. In the centralized locking algorithm, the new transaction is delayed until the blocking transaction terminates. In the distributed locking algorithm, either the new transaction is delayed until the blocking transaction terminates or the blocking transaction is preempted by the new transaction.

Chapter 7

Simulation Results

The simulation results are presented and analyzed in this chapter. The results have been obtained using both discrete event simulation and EPTN simulation and are also reported in [KoO86].

7.1. Effect of Interarrival Time

Figure 7.1 shows the effect of mean interarrival time on the mean response time of transactions. We notice that in most cases, except under very heavy load (that is, for $I_t < 5$ secs), the CL algorithm performs better than the DL algorithm. The higher response time obtained for the DL algorithm is due mainly to the extra delays that are incurred by (1) having to wait for locks to be granted by all sites , and (2) having to wait for locks to be released locally before the transaction could be terminated. Recall that in the CL algorithm locks are granted by one site only and that transactions do not have to wait for their locks to be released before they could complete unless they originated at the central site. When I_t decreases below 5 seconds there is a more sudden upturn in the CL curve than in the DL curve. The sharper increase in response time for the CL algorithm is caused by the higher demand on the I/O resources at the central site as the system becomes saturated with transactions (bottleneck effect).

This is depicted in Figure 7.2. The CPU utilization, presented in Figure 7.3, shows a similar behavior to the I/O utilization. However, the CPU utilization is so low compared with the I/O utilization that its effect on the mean response time is negligible.

The average number of messages per update transaction as a function of interarrival time is presented in Figure 7.4. The horizontal lines obtained suggest that the mean number of messages for both algorithms is independent of the load. The extra messages for the DL algorithm are due to the additional lock requests that are sent to all sites. Recall that we assume a point-to-point type of network for the network model, which means that broadcasting a message to N sites is equivalent to sending N separate messages. For a broadcast type of network, we expect the results for both algorithms be very similar since the same rounds of messages would be exchanged before a transaction is committed.

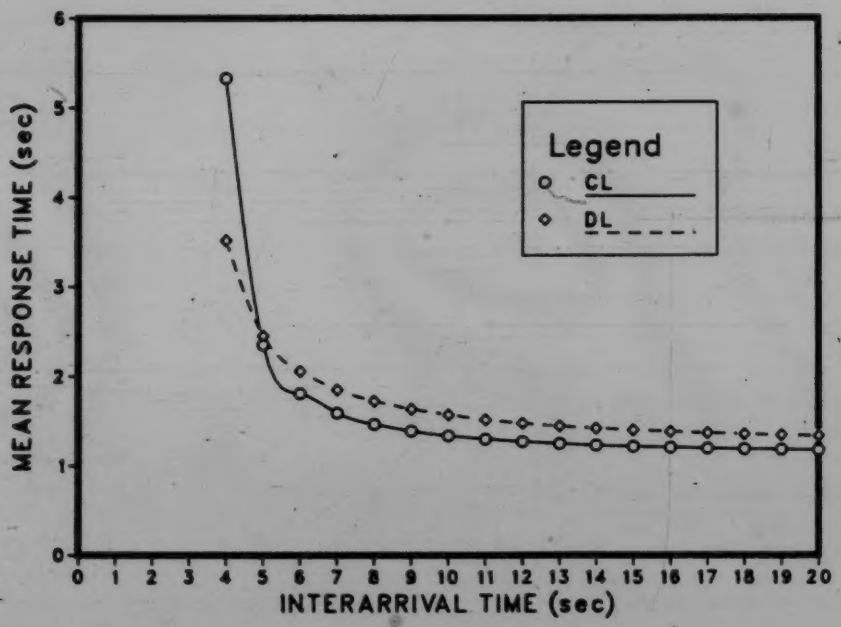

Figure 7.1. Effect of mean interarrival time on mean response time.

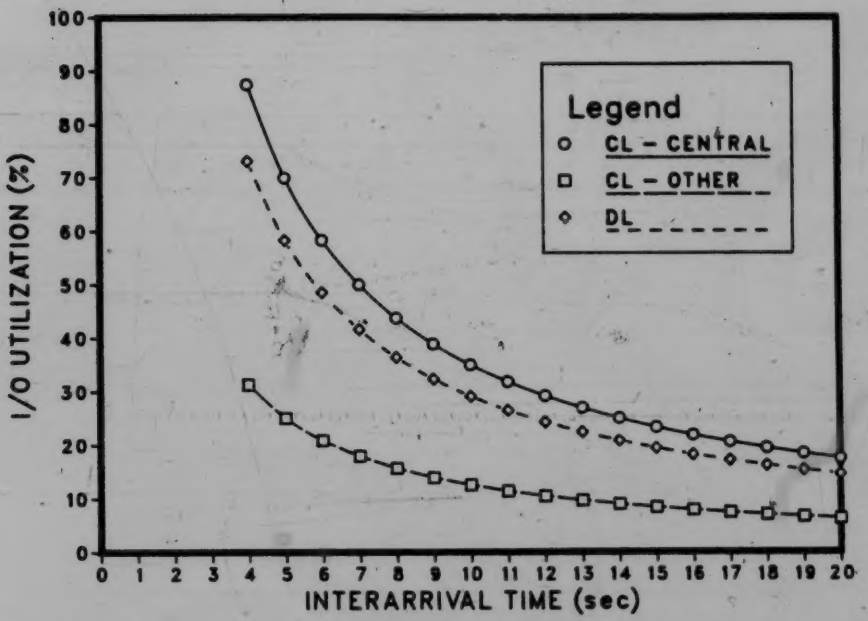

Figure 7.2. Effect of mean interarrival time on I/O utilization.

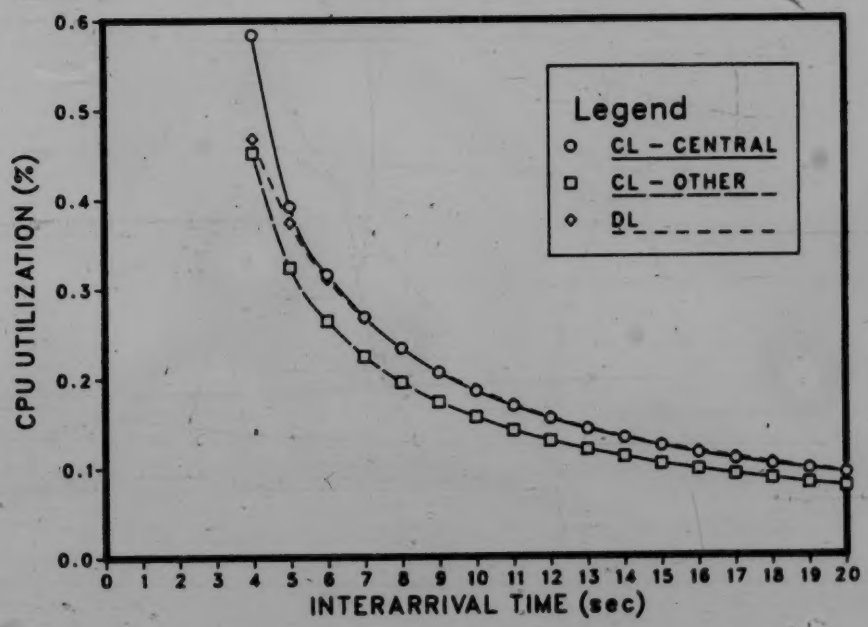

Figure 7.3. Effect of mean interarrival time on CPU utilization.

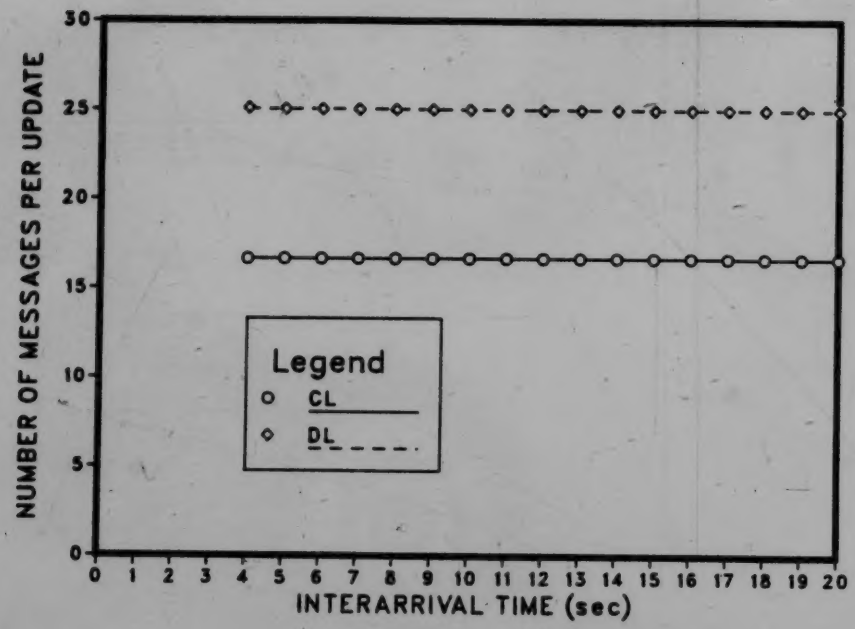

Figure 7.4. Effect of mean interarrival time on mean number of messages.

7.2. Effect of Base-set Size

The effect of base-set size on the mean response time of transactions is shown in Figure 7.5. Increasing the base-set size causes an increase in the mean response time for both algorithms. For CL the increase is linear until the base-set size reaches 5 items where the rate of increase starts to rise rapidly. For DL the change in the rate of increase occurs at a higher base-set size. The increase in mean response time is caused mainly by the increase in I/O utilization, shown in Figure 7.6, and the longer delays that are introduced due to the increase in number of conflicts among transactions. The rapid increase in response time is accounted for by the fact that the delays due to conflicts are negligible when the base-set size is small, but become more dominant as the base-set size gets large.

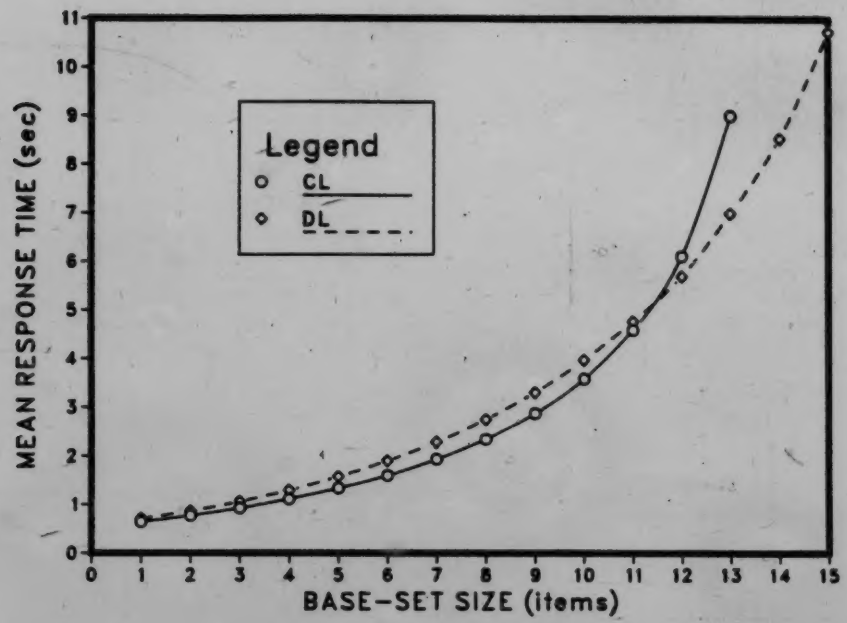

Figure 7.5. Effect of mean base-set size on mean response time.

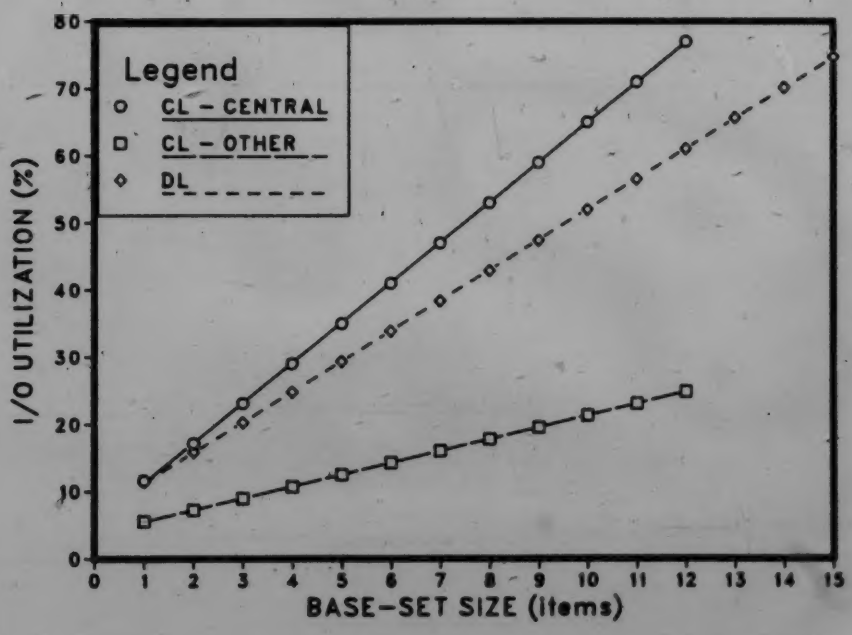

Figure 7.6. Effect of mean base-set size on I/O utilization.

7.3. Effect of Number of Sites

The effect of the number of sites on the performance of the algorithms in terms of mean response time of transactions is shown in Figure 7.7. For both algorithms, increasing the number of sites in the system degrades the performance. For the CL algorithm, the mean response time increases slowly at first until N reaches about 12 sites where it starts to rise rapidly. For the DL algorithm, the response time increases steadily. In both cases, the increase in response time is due to the fact that as the number of sites increases the overall rate of arrival of transactions into the system also increases causing additional delays on the processing of the transactions. The sharp increase in the CL curve occurs as a result of the high demand on the I/O resource at the central site which is swamped by transactions (Figure 7.8). For the DL algorithm, the increase is not as dramatic since the load is distributed among the sites.

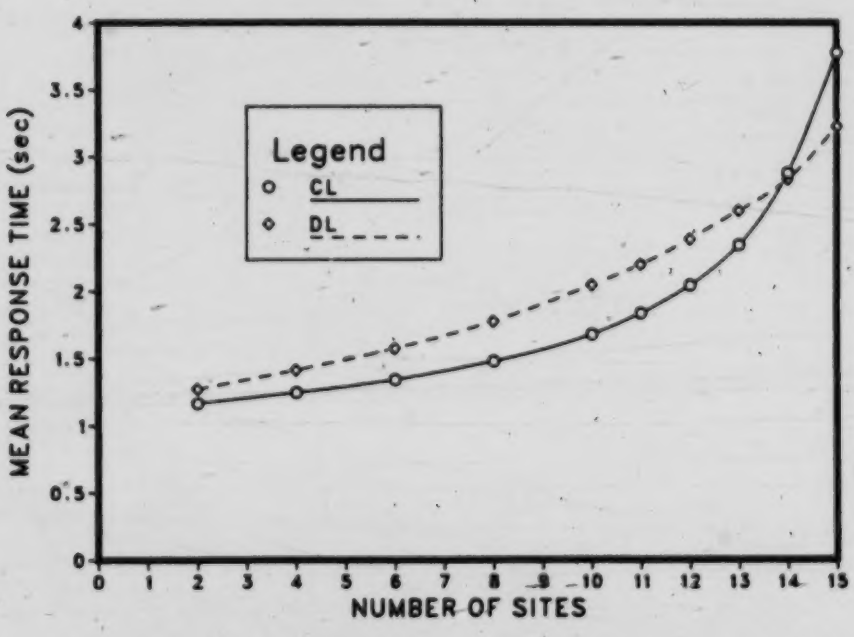

Figure 7.7. Effect of number of sites on mean response time.

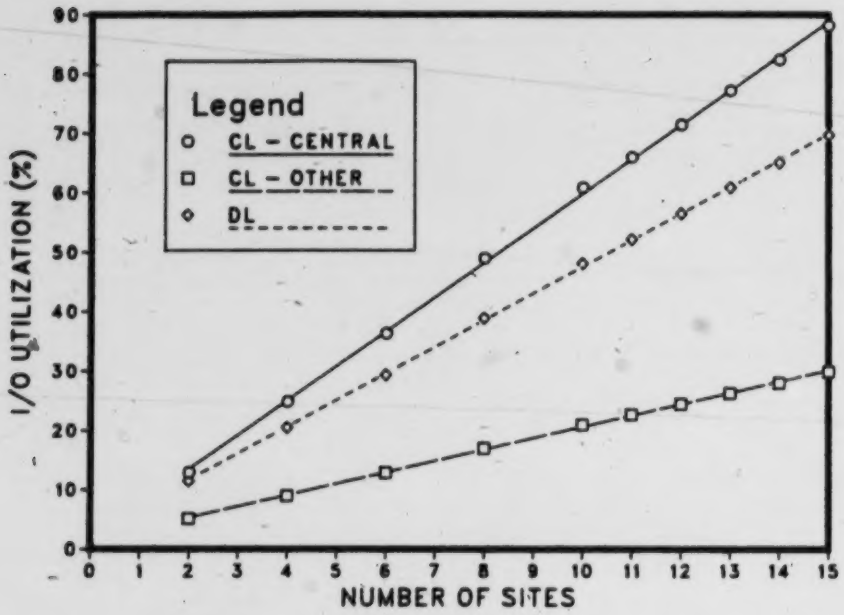

Figure 7.8. Effect of number of sites on I/O utilization.

7.4. Effect of Transmission Time

The effect of the transmission time, T, on the mean response time is depicted in Figure 7.9. An increase in transmission delays affects the DL algorithm more than the CL algorithm. This is reasonable since on the average more messages are exchanged between sites in the DL algorithm than in the CL algorithm (see Figure 7.4).

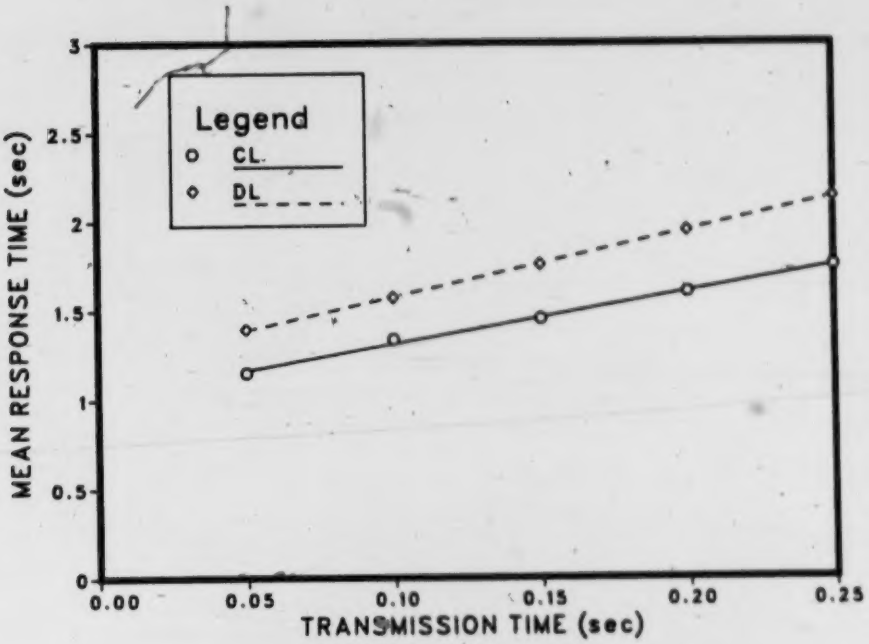

Figure 7.9. Effect of transmission time on mean response time.

7.5. Effect of I/O Synchronization Time

The I/O synchronization time is the time it takes to read/write an item lock. An I/O synchronization time of 0 seconds represents a system where the lock tables are kept in main memory. The effect of this parameter on the mean response time is shown in Figure 7.10. There is a considerable improvement in the performance of both algorithms as the I/O synchronization time is decreased. This result is very important for distributed database designers. It indicates that better performance could be achieved by reducing the overhead due to I/O synchronization to a minimum.

The effect of the I/O synchronization time on the I/O utilization is shown in Figure 7.11. One very interesting observation is that the I/O utilizations at the central site and the other sites for the CL algorithm converge to the same value as the I/O synchronization approaches zero. The explanation for this is simple. The only I/O processing that is done at the central site and is not done at the other sites is the

locking and unlocking of items; when the cost for this is reduced to zero, the I/O utilization at the central site should be equal to that at the other sites. One interesting conclusion is: in a system where the I/O synchronization time is equal to zero, the bottleneck effect (see sections 7.1 and 7.2) caused primarily by the higher I/O utilization at the central site would not occur. To verify this, we set the I/O synchronization time to zero and ran the simulation for the two algorithms under different loads. As shown in figures 7.12 and 7.13, the results confirm what we anticipated. Notice that the CL curve overlaps the DL curve indicating that the I/O utilization for reading and updating data items is the same for both algorithms.

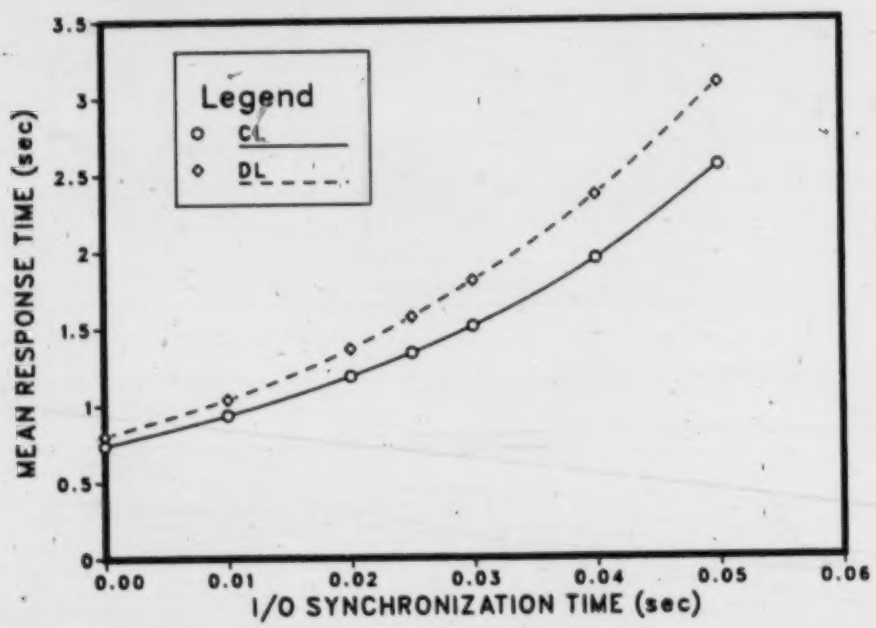

Figure 7.10. Effect of I/O synchronization time on mean response time.

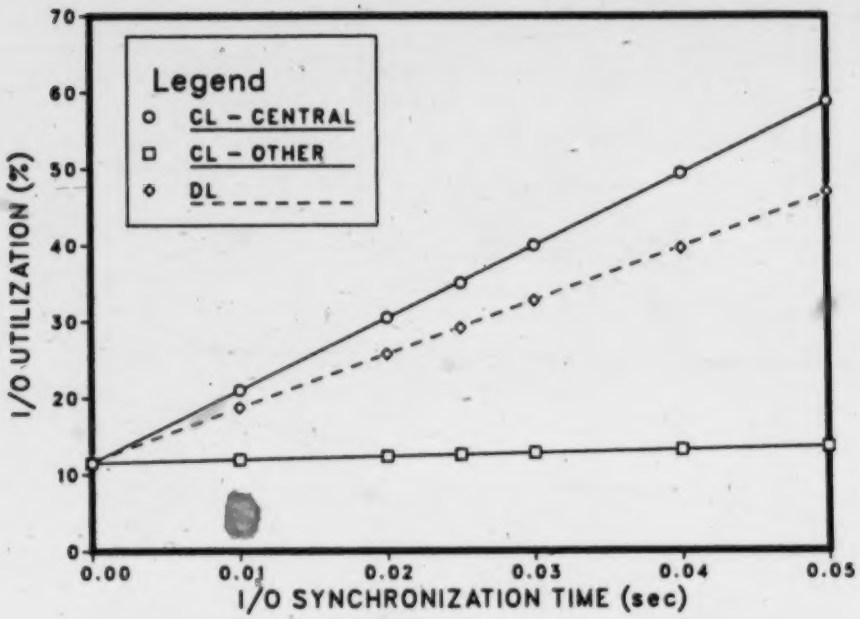

Figure 7.11. Effect of I/O synchronization time on I/O utilization.

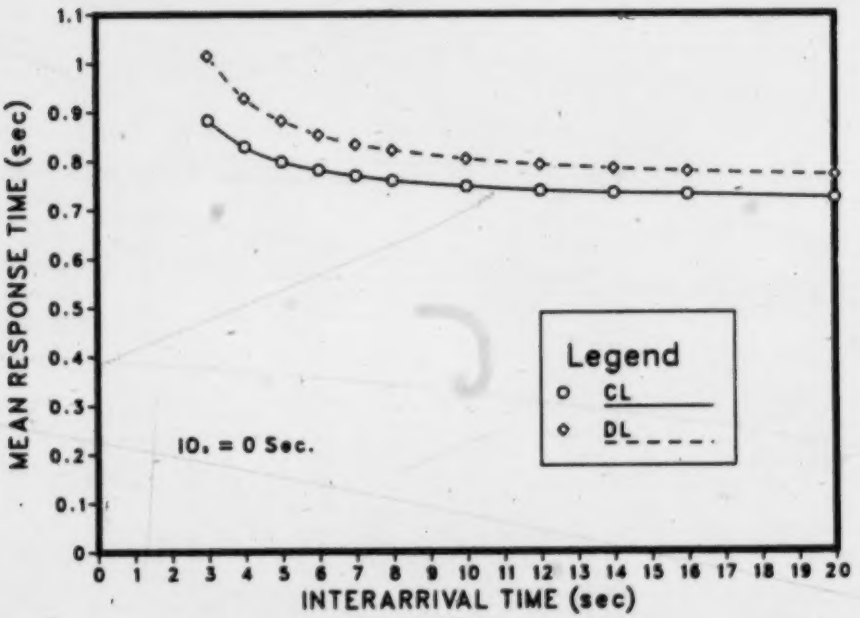

Figure 7.12. Effect of interarrival time on response time ($IO_s = 0$).

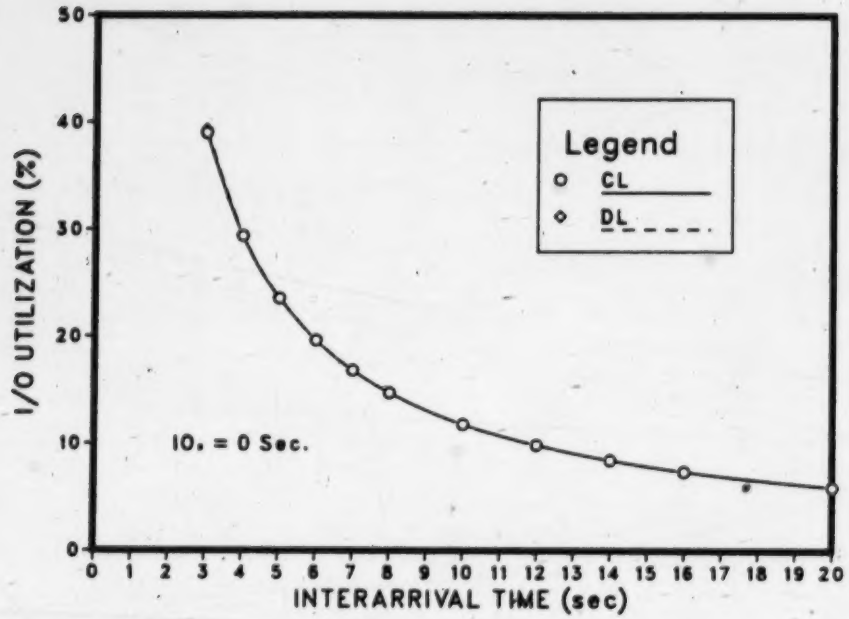

Figure 7.13. Effect of interarrival time on I/O utilization ($IO_s = 0$).

7.6. Effect of Database Size

Figure 7.14 shows the effect of the database size on the mean response time. For large databases the response time for both the DL and the CL algorithms is constant. The DL curve shows a higher mean response time as expected. As the number of items in the database is reduced below 100, the mean response time increases very rapidly. This is caused by the larger number of conflicts that occur when the transactions have to compete for a smaller number of items.

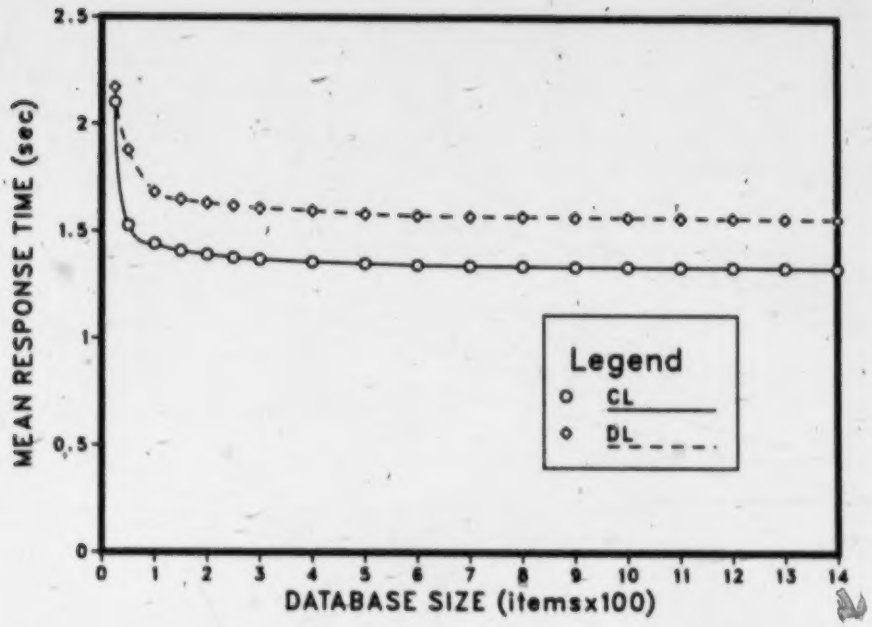

Figure 7.14. Effect of database size on mean response time.

7.7. Effect of Mean Time Between Failures

Figure 7.15 shows the relationship between the interarrival time, the mean time between failures and the mean response time. In most cases, the centralized locking algorithm performs better than the distributed version. As the mean time between failures is decreased, the upturn in the curves becomes less sharp. This is due to the decrease in I/O utilization, as shown in Figure 7.16, which results from a decrease in the total number of transactions that are being received and processed by the functional sites. Notice that at large interarrival times, the response time is not affected much by a change in the failure rate since that time represents the minimum average time that a transaction takes to complete.

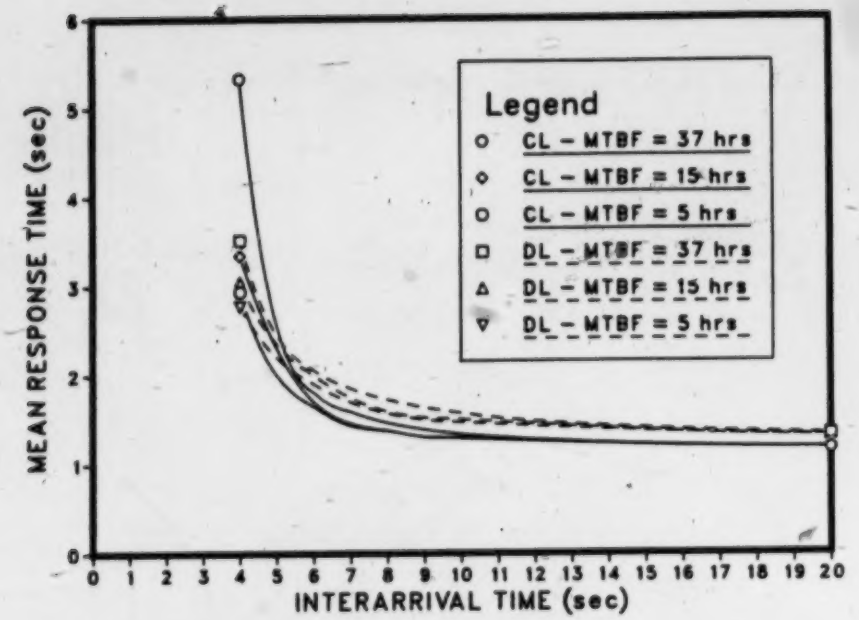
MEAN RESPONSE TIME (sec)
INTERARRIVAL TIME (sec)
Legend
CL – MTBF = 37 hrs
CL – MTBF = 15 hrs
CL – MTBF = 5 hrs
DL – MTBF = 37 hrs
DL – MTBF = 15 hrs
DL – MTBF = 5 hrs

Figure 7.15. Effect of mean time between failures on response time.

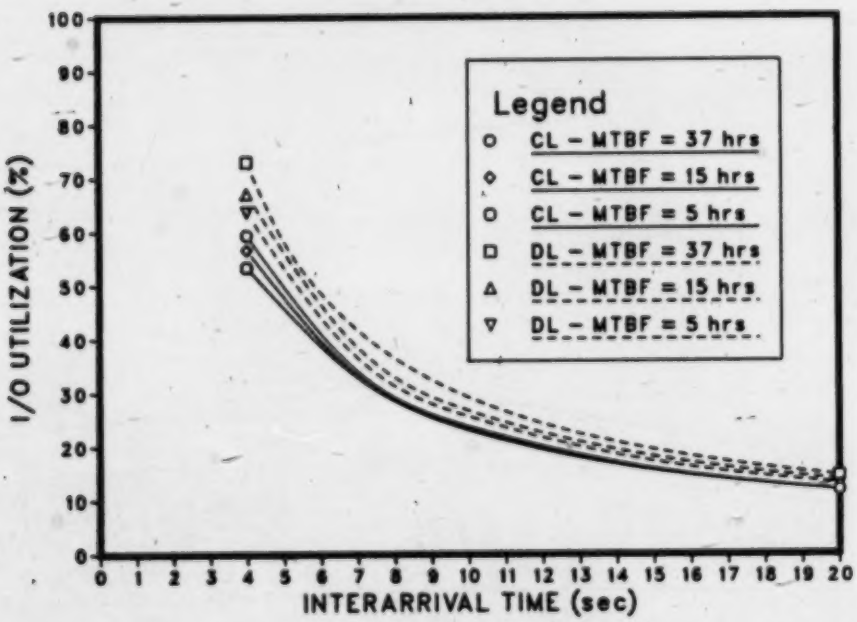
I/O UTILIZATION (%)
INTERARRIVAL TIME (sec)
Legend
CL – MTBF = 37 hrs
CL – MTBF = 15 hrs
CL – MTBF = 5 hrs
DL – MTBF = 37 hrs
DL – MTBF = 15 hrs
DL – MTBF = 5 hrs

Figure 7.16. Effect of mean time between failures on I/O utilization.

The effect of mean time between failures on the availability of the centralized locking algorithm is shown in the table below. Contrary to what we expected, the availability of the CL algorithm is extremely high. This is explained by the fact that although the system is vulnerable to central site failures, the availability of the entire system is not affected much since the time it takes to elect a new central site and to resume normal operation is usually very short.

Table 7.1. Effect of mean time between failures on availability.

MTBF (Hours)	Availability (%)	
	CL	DL
1	97.86	98.16
2	99.98	100.0
3	99.99	100.0
4	99.99	100.0
5	99.99	100.0
10	99.99	100.0
15	99.99	100.0

7.8. EPTN Simulation Versus Discrete Event Simulation

In the introduction, we mentioned that one of the objectives of this study is to show the appropriateness of the EPTN formalism as a performance modeling tool. To achieve this, some of the discrete event simulation (DES) experiments were repeated using EPTN simulation. The results and complexities, in terms of ease of implementation and simulation cost, are compared in this section. Our experiences in using the tool and plans for future improvements are also presented.

The results obtained from both DES and EPTN simulation for different interarrival times are listed in the tables below. As can easily be deduced from the values, the results are very similar.

With regard to the ease of implementation, implementing a system using DES is far more complex and difficult than using EPTN and SIMNET. This is particularly

Table 7.2. Comparison of EPTN and DES. Effect of interarrival time on mean response time.

Interarrival Time	Mean Response Time			
	CL		DL	
	DES	EPTN	DES	EPTN
4	5.332	5.233	3.518	3.732
5	2.350	2.414	2.451	2.748
6	1.807	2.010	2.055	2.090
7	1.590	1.601	1.849	1.948
8	1.465	1.502	1.723	1.812
9	1.389	1.462	1.635	1.691
10	1.335	1.384	1.570	1.558
12	1.270	1.335	1.476	1.525
14	1.228	1.289	1.420	1.474
16	1.203	1.267	1.382	1.438
18	1.185	1.242	1.352	1.379
20	1.170	1.216	1.331	1.335

Table 7.3. Comparison of EPTN and DES. Effect of interarrival time on I/O utilization for CL algorithm.

Interarrival Time	I/O Utilization			
	DES		EPTN	
	Central Site	Other Sites	Central Site	Other Sites
4	87.445	31.331	89.264	31.966
5	69.965	25.070	69.985	25.786
6	58.305	20.893	59.988	21.480
7	49.979	17.909	49.952	17.239
8	43.735	15.671	44.953	16.103
9	38.875	13.928	39.958	14.314
10	34.988	12.536	35.937	12.882
12	29.157	10.447	29.935	10.735
14	24.992	8.955	25.660	9.196
16	21.868	7.835	22.448	8.043
18	19.438	6.965	19.938	7.146
20	17.494	6.268	17.934	6.430

Table 7.4. Comparison of EPTN and DES. Effect of interarrival time on I/O utilization for DL algorithm.

Interarrival Time	I/O Utilization	
	DES	EPTN
	All Sites	All Sites
4	73.100	72.677
5	58.322	58.387
6	48.555	48.631
7	41.608	41.547
8	36.402	36.453
9	32.362	32.346
10	29.112	29.149
12	24.256	24.226
14	20.791	20.760
16	18.188	18.203
18	16.167	16.163
20	14.551	14.550

true when the system contains asynchronous and concurrent activities since the basic mechanisms for controlling the interactions among concurrent activities need to be implemented in DES; whereas, in EPTN these are embedded in the net structure in the way the transitions and places are linked together. As an indication of the complexity involved, the amount of code and amount of work required in both cases are compared. Using DES with SIMULA as the base language, the CL algorithm only was written in approximately 1500 lines of code and took about 2 months to complete, while using EPTN as the modeling formalism and SIMNET as the implementation language, both the CL and DL algorithms were implemented in less than 900 lines of code and took less than 2 weeks to complete! However, I should point out that I implemented the algorithms in SIMULA before implementing them in SIMNET; therefore, part of the large difference in time might be due to my inexperience with SIMULA and the algorithms initially.

As for the simulation cost, EPTN simulation is about two to three times more expensive to run than DES. This is mainly due to the fact that, in EPTN simulation,

every time a token is deposited in a place, the precondition for all the output transitions of that place should be checked for trueness. This process usually involves several procedure calls and is very expensive in SIMULA. However, work to improve the simulator is under way. We are currently planning either to implement a preprocessor that would attempt to optimize the code before a simulation is run or to implement SIMNET in a "lower level" language, such as C, which probably would give us more control over the primitive operations.

Chapter 8

Conclusion

In this thesis, we have used two different methodologies to study the performance of two resilient concurrency control algorithms for distributed databases in the presence of site failures. The first methodology is based on traditional discrete-event simulation (DES), and the second one is a new methodology based on the simulation of systems modeled as extended place/transition nets (EPTN). In doing so, we have made the following contributions.

We have investigated the problems of maintaining consistency and reliability in distributed databases, and we have outlined, in some detail, several of the alternatives that are available to the designer of reliable distributed databases.

We have developed and implemented a simulator (called SIMNET) for the EPTN formalism. The simulator could be used for obtaining performance results for distributed systems in general and distributed systems in particular. We have experimented with the simulator, and have found that while the results obtained are very similar to those obtained using DES, the time and effort required are much less.

We have studied two very popular concurrency control algorithms for distributed databases, namely the centralized locking algorithm (CL) and the distributed locking

algorithm (DL), and we have reached several new conclusions concerning the two algorithms. Earlier work by Garcia-Molina [Gar79] and Ozsu [Ozs85b] has shown that the CL algorithm performs much better than the DL algorithm in a failure-free environment. However, because of the vulnerability of the scheme to central site failures there were some doubts regarding the performance in environments with failures. More recent work by Cheng [Che81] has tried to use a backup central site to improve the reliability of the scheme but has found that the resulting scheme performs much worse than a distributed scheme. In this study, we have taken another approach. Instead of using a backup site, we have chosen to use an election protocol, as suggested by Garcia-Molina [Gar79] in a resilient version of the CL algorithm, to rapidly elect a new central site and to recover from a central site crash. The results we have obtained are quite surprising. We have found that:

1. The availability of the CL algorithm is quite high and is comparable to that of the DL algorithm. This result is reasonable if we consider the fact that central site failures do not occur very often, and the time it normally takes to elect a new central site and to recover from a failure in case of a central site failure is relatively small.

2. The CL algorithm does not incur any additional overhead during normal operation since it is only during central site failures that the election protocol is invoked.

3. The CL algorithm still outperforms the DL algorithm in most cases. The performance of the CL algorithm deteriorates only under very heavy load, since the demand on the I/O resource at the central site is heavy in that circumstance (bottleneck effect). However, if the I/O synchronization time is reduced to zero by keeping the lock table in main memory, the CL algorithm outperforms the DL algorithm in any type of situation. Thus, the bottleneck effect in the CL

algorithm is not a major problem since it can be alleviated by keeping the lock table in main memory for fast access.

8.1. Suggestions for Future Research

There are several extensions that could be made to the work presented here.

1. The user interface for the net simulator could be extended to incorporate a graphics-based net editor. The editor would allow the control structures of the models to be "drawn" rather than described. This would considerably increase the ease-of-use of the tool. Some of the information, for example, the transition procedures, would still have to be specified textually, but the major portion of the description could be done graphically. The tracing of the net execution could also be animated by showing the firing of transitions and the movement of tokens on a graphical display of the net.

2. Another area that needs further research involves the development of analytical solution for EPTN. Some work has already been done in this area for some restricted types of Petri nets ([Sif77], [Mol82]). However, analytical solutions for general net structures have yet to be developed.

3. Another direction of research is to remove some of the simplifying assumptions made in this study. This perhaps would provide more realistic results. For example, the network model could be improved to reflect the characteristics of some existing communication network systems. It would also be interesting to analyze the performance of the algorithms on a broadcast-type of network.

4. In this study, we studied only locking-based concurrency control algorithms. This research could be extended to encompass other types of algorithms, such as, timestamp-based and optimistic methods.

Bibliography

[Age78] T. Agerwala, " Putting Petri Nets to Work", *IEEE computer 12*, 12 (Dec. 1978), 85-94.

[AlD76] P.A. Alsberg and J.D. Day, " A Principle for Resilient Sharing of Distributed Resources", *Proc. Second Intl. Conf. on Software Engineering*, Oct. 1976.

[AHY83] P.M.G Apers, A.R. Hevner and S.B. Yao, " Optimization Algorithms for Distributed Queries", *IEEE Trans. on Software Eng. SE-9*, 1 (1983), .

[Asa79] M.M. Astrahan and *et al.*, " System R: A Relational Data Base Management System", *Computer 12*, 5 (May 1979), 47-48.

[BaP78] D.Z. Badal and G.T. Popek, " A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Database Systems" , *Proc. Third Berkeley Workshop on Distrbuted Data Management and Computer Networks*, Aug. 29-31, 1978, 273-288.

[BeG81] P.A. Bernstein and N. Goodman, " Concurrency Control in Distributed Database Systems", *Computing Surveys 13*, 2 (June 1981), 185-222.

[Bir81a] G. Birtwistle, " Introduction to Demos", *IEEE Winter Simulation Conference Proceeding*, 1981, 559-571.

[Bir81b] G. Birtwistle, *Demos Implementation Guide and Reference Manual*, Research Report No. 81/70/22, University of Calgary, Nov. 1981.

[CeP84] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw Hill Inc., 1984.

[ChA80] P.P.S. Chen and J. Akoka, " Optimal Design of Distributed Information Systems", *IEEE Trans. on Computers C-29*, 12 (1980), :

[Che81] W.K. Cheng, *Performance Analysis of Update Synchronization Algorithms for Distributed Databases*, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1981.

[ChO74] W.W. Chu and G. Ohlmacher, " Avoiding Deadlock in Distributed Databases", *ACM National Symposium*, 1974, 156-160.

[ChN75] W.W. Chu and E.E. Nahoraii, " File Directory Design Considerations for Distributed Databases", *First Int'l Conf. on Very Large Databases*, 1975, 543-545.

[DaN66] O. Dahl and K. Nygaard, " SIMULA - An Algol-Based Simulation Language", *Comm. ACM 9*, 9 (1966), 671-687.

[DMN84] O. Dahl, B. Myhrhaug and K. Nygaard, *SIMULA 67 - Common Base Language*, Norwegian Computing Center, Feb. 1984.

[Ell77] C.A. Ellis, " A Robust Algorithm for Updating Duplicated Databases", *Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks*, May 25-27, 1977, 146-158.

[Fra77] W.R. Franta, *The Process view of Simulation*, North-Holland Inc., New-York, 1977.

[Gar79] H. Garcia-Molina, *Performance of Update Algorithms for Replicated Data in a Distributed Database*, Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, California, 1979.

[Gar80] H. Garcia-Molina, " Reliability Issues For Completely Replicated Distributed Databases", *Proc. COMPCON*, 1980, 442-449.

[Gar82] H. Garcia-Molina, " Elections in a Distributed Computing Systems", *IEEE Trans. on Computers C-31*, 1 (Jan. 1982), 48-59.

[GaC80] G. Gardarin and W.W. Chu, " A Distributed Control Algorithm for Reliably and Consistently Updating Replicated Databases", *IEEE Trans. on Computers C-29*, 12 (Dec. 80), 1060-1068.

[GeL79] H.J. Genrich and K. Lautenbach, " The Analysis of Distributed Systems by Means of Predicate/Transition Nets", in *Semantics of Concurrent Computation*, G. Kahn (ed.), Springer-Verlag, 1979, 123-146.

[Gor78] G. Gordon, *System Simulation*, Prentice Hall, 1978.

[Gra78] J.N. Gray, Notes on Database Operating Systems, in *Operating Systems, An Advanced Course*, R. Bayer, R.M. Graham and G. Seegmuller (ed.), Springer-Verlag, New-York, 1978, 393-481.

[HoC70] A.W. Holt and T. Commoner, " Events and Conditions", *Record of the Project Mac Conference on Concurrent Systems and Parallel Computation*, 1970, 3-52.

[Koh81] W.H. Kohler, " A Survey of Techniques for Synchronization on Recovery in Decentralized Computer Systems", *Computing Surveys 13*, 2 (June 1981), 150-183.

[KoO86] T.M. Koon and M.T. Ozsu, " Performance Comparison of Resilient Concurrency Control Algorithms for Distributed Databases", *Proc. 2nd*

International Conference on Data Engineering, Feb. 4-6, 1986, (accepted for publication).

[Lam78] L. Lamport, " Time, Clocks and the Ordering of Events in a Distributed System", *Comm. ACM 21*, 7 (July 1978), 558-565.

[Lam83] G. Lamprecht, *Introduction to SIMULA 67*, Fried. Vieweg & Sohn Velagsgesellschaft mbH, Braunschweig , 1983.

[LaS76] B. Lampson and H. Sturgis, " Crash Recovery in Distributed Data Storage System", *Technical Report, Xeroz Palo Alto Res. Center*, Palo Alto, Ca., 1976.

[Lan78] G. Le Lann , " Algorithms for Distributed Data Sharing Systems Which use Tickets", *Proc. Third Berkeley Workshop on Distributed Data Management and Computer Networks*, 1978, 259-272.

[LiN83] W.K. Lin and J. Nolte, " Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking", *Proc. 9th International Conf. on Very Large Data Bases*, 1983, 109-119.

[Mal80] T.A. Marsland and S.S. Isloor, " Detection of Deadlocks in Distributed Database Systems" , *INFOR 18*, 1 (Feb. 1980), 1-20.

[Mol82] M.K. Molloy, " Performance Analysis Using Stochastic Petri nets", *IEEE Transaction on Computers C-31*, (Sep. 1982), 913-917.

[Noe79] J.D. Noe, " Nets in Modelling and Simulation", in *Net Theory and Applications*, Springer-Verlag, New York, 1979, 347-368.

[Nut72] G.J. Nutt, *The Formulation and Application of Evaluation Nets*, Ph.D. Thesis, University of Washington, Seattle, Washington, 1972.

[Obe82] R. Obermack, " Distributed Deadlock Detection Algorithm", *ACM Trans. Database Systems 7*, 2 (June 1982), 187-208.

[Ozs82] M.T. Ozsu, " An Introduction to Distributed Databases", *Proc. 2nd Regional Seminar on Microprocessors/Microcomputers and Distributed Computer Systems*, Ankara, Turkey, Oct. 1982, 304-319.

[Ozs85a] M.T. Ozsu, " Modeling and Analysis of Distributed Database Concurrency Control Algorithm Using an Extended Petri Net Formalism", *IEEE Trans. on Software Eng. SE-11*, 10 (1985), 1225-1240.

[Ozs85b] M.T. Ozsu, " Performance Comparison of Distributed vs Centralized Locking Algorithms in Distributed Database Systems", *Proc. 5th International Conference on Distributed Computing Systems*, May 1985.

[Pap79] C.H. Papadimitriou, " The Serializability of Database Updates", *J. ACM 26*, 4 (Oct. 1979), 631-653.

[Pet81] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[Pet62] C. Petri, *Kommunikation mit Automaten*, Ph.D. Thesis, University of Bonn, Bonn, Federal Republic of Germany, 1962.

[Rie79] D.R. Ries, *The Effects of Concurrency Control on Database Management System Performance*, Ph.D. Thesis, Department of EECS, University of California at Berkeley, Berkeley, California, 1979.

[SaY82] G.M. Sacco and S.B. Yao, " Query Optimization in Distributed Database Systems", in *Advances in Computers*, vol. 21, Academic Press, New York, 1982, 225-273.

[SeL76] D.G. Severance and G.M. Lohman, " Differential Files: Their Application to the Maintenance of Large Databases", *ACM Trans. Database Systems 1*,

(Sep. 1976), 256-261.

[Sif77] J. Sifakis, Use of Petri Nets for Performance Evaluation, in *Measuring, Modelling and Evaluating Computer Systems*, H. Beilner and E. Gelenbe (ed.), North-Holland Publishing Co., 1977, 75-93.

[Ske81] D. Skeen, " Non-Blocking Commit Protocols", *Proc. ACM/SIGMOD International Conference on Management of Data*, Ann Arbor, Michigan, 1981, 133-142.

[Tho79] R.H. Thomas, " A Majority Consensus Approach to Concurrency Control ", *ACM Trans. Database Systems 6*, 2 (1979), 180-209.

[Ver78] J.S. Verhofstad, " Recovery Techniques for Database Systems", *Computing Surveys 10*, (June 1978), 168-195.

Appendix 1

SIMNET User's Manual

This manual describes the use of the SIMNET simulation package available under the Michigan Terminal System at the University of Alberta.

In the description that follows, SIMNET reserved words will be set in *italics* and SIMULA keywords will be set in **bold**, and user-defined identifiers will be set in UPPER CASE. Angle-brackets ("<" and ">") will be used to represent portions of SIMNET statements that are to be replaced by appropriate user constructs. For example, <identifier> indicates that the user is to insert an identifier at this place. Square brackets will be used to indicate a portion of a statement that is optional. Vertical bars ("|") will be used to separate alternative productions. Any syntactic units referred to, but not defined in this manual, refer to the syntactic definitions given in the SIMULA Common Base Language manual [DMN84]. Construction of the form <SIMULA some syntactic class> stands for the syntactic definition of <some syntactic class> given in [DMN84].

1. Invoking SIMNET

The SIMNET context is invoked as an external class by placing the following statement after the beginning of the program but before the prefixed block as follows:

```
begin
  external class SIMNET=KOOM:SIMNET;
    ....
    SIMNET begin
    ....
```

To compile the program the following MTS command could be used:

RUN *SIMULA SCARDS=<source program> PAR=NOWARN

The parameter NOWARN suppresses the printing of the warning messages which can be ignored when running a SIMNET program!

If there is no syntax errors in the program, the object code will be placed in the temporary file -SIMLOAD. The object program could then be run as follows:

RUN -SIMLOAD+*SIMLIB

For more information regarding the MTS SIMULA compiler, refer to the SIMULA reference guide R475.0884 released by the Department of Computing Services.

2. Net Model Construction

This section of the manual describes the use of the predefined SIMNET classes and procedures to declare and construct a simulation model from an Extended Place/Transition net. Each class is described according to the following format:

Class class-name

A brief description of the class.

Outline:

Any attributes of the class that the user needs to know will be outlined here.

Subclass Declaration Syntax:

If a subclass of the class should be declared before it could be used, the syntax for the subclass declaration will be given here.

Example:

A complete example on how to use the class will be presented here.

3. Class Dataset

The SIMNET class dataset could be used to declare the data objects to be carried by data tokens.

Outline:

```
class dataset;
   begin
      ref(subnet) snet;
      integer num;
      ....
   end;
```

Snet refers to the subnet where the dataset originated. *Num* represents a unique identification number that has been assigned to the data object.

Subclass Declaration Syntax:

```
dataset class <data object name> [(<formal parameter part>);
   <specification part>];
   begin
      <SIMULA class body>
   end;

<data object name> ::= <identifier>
```

Example:

In this example, we illustrate how to use the dataset class to declare data objects that would represent, let's say, transactions in a database system. Assume that each transaction should have the following attributes:

1. TIMEIN - the time that the transaction enters the system,

2. BASESET - the number of items being referenced by the transaction, and

3. WRITESET - the number of items being modified by the transaction.

Also, assume that the value for BASESET is to be obtained at the moment of transaction generation from a negative exponential distribution with a mean value (MEAN) which is passed as a parameter, and that WRITESET is equal to the greatest integer less than or equal to BASESET divided by 2. The transaction class declaration in SIMNET is as follows.

```
dataset class TRANSACTION(MEAN);
   integer MEAN;
      begin
      integer BASESET, WRITESET;
      real TIMEIN;

      TIMEIN := Time;
      BASESET := Negexp(1/MEAN, seed1);
      WRITESET := Entier((BASESET+1)/2);
   end;
```

Time is a SIMULATION built-in function returning the current simulation time. Negexp is a SIMULA random number generator returning a drawing from a negative exponential distribution (see Section 13.2). *Seed1* is a SIMNET global integer variable containing a predefined seed value (see Section 13.1).

4. Class Source

This facility could be used to automatically generate new data tokens for the net simulation according to a given stochastic function.

Outline:

```
class source;
   virtual ref(dataset) procedure newdata;
         real procedure dist;
   begin
```

```
        integer obs;
        real resetat;
        ....
        procedure connect(pl);
            ref(place) pl;
            begin
                ....
            end;
        ....
    end;
```

Newdata and *dist* are two procedures that need to be redefined in a subclass of the class source if this facility is to be used. *Newdata* specifies the type of data object that should be generated and *dist* specifies the type of distribution that should be used. *Obs* and *resetat* keep track of some performance statistics about this facility. *Obs* keeps track of the total number of data objects that have been generated since the last reset time. The last reset time is kept in *resetat* and represents the time the source was last reset. The procedure *connect* is used to connect the source to the place that will be receiving the data tokens.

Subclass Declaration Syntax:

```
class source <source name>;
    begin
        ref(dataset) procedure newdata;
        newdata :- new <data object name>;

        real procedure dist;
            dist := <stochastic function>;
    end;

<source name> ::= <identifier>
<stochastic function> ::=
    Draw(<real>, <seed>) | Negexp(<real>, <seed>) |
    Normal(<real>, <seed>) | Randint(<real>, <seed>) |
    Poisson(<real>, <seed>) | Erlang(<real>, <real>, <seed>) |
    Uniform(<real>, <seed>) | Histd(<array>, <seed>) |
    Discrete(<array>, <seed>) | Linear(<array>, <array>, <seed>)
<real> ::= <real value expression>
<array> ::= <array identifier>
<seed> ::= seed1 | seed2 | seed3 | seed4 | seed5 | seed6 | seed7 |
    seed8 | seed9 | seed10 | <integer variable identifier>
```

See Section 13 for more details about seeds and stochastic functions.

Example:

The following example shows how to declare a source NEWSOURCE that would generate transactions at interarrival times obtained from a negative exponential distribution with a mean MIT. The new data tokens are to be deposited in a place referenced by variable PL.

```
ref(place) PL;
....
source class NEWSOURCE;
   begin
      ref(dataset) procedure newdata;
         newdata :- new TRANSACTION;

      real procedure dist;
         dist := Negexp(1/MIT, seed1);
   end;
```

Later on in the program, a new source is created and connected to PL by:

```
new NEWSOURCE.connect(PL);
```

5. Class Sink

Class *sink* is a facility provided by SIMNET to absorb used data tokens from a place in the net.

Outline:

```
class sink;
   begin
      integer obs;
      real resetat, sum, sumsq;
      ....
   end;
```

Obs holds the number of data tokens that have been absorbed since the last reset time *resetat*. *Sum* and *sumsq* store the sum and the sum of the squares of the times spent in the system by data objects that originated from a source in the same subnet since *time* = *resetat*.

Example:

The following example illustrates how a sink is used to absorb tokens from a place referenced by, let's say, variable PL (see Section 9 also).

```
ref(place) PL;
....
new sink.connect(PL);
```

8. Class Resource

Class *resource* is a facility that is provided by SIMNET to keep track of performance related statistics, such as, usage, queue length, throughput, etc.

Outline:

```
class resource(nounits, nam);
    value nam;
    text nam;
    integer nounits;
    begin
        ....
        real procedure usage; ....
        real procedure tput; ....
        real procedure qtime; ....
        real procedure qlength; ....
        real procedure qsd; ....
        ....
        procedure connect(pl);
            ref(place) pl;
            begin
                ....
            end;
        ....
    end;
```

Nounits is an integer parameter specifying the number of units of the resource to be created. *Nam* is a text value parameter representing the name of the resource. The name is used by the tracing facility and the report routine.

The locally defined functions *usage, tput, qtime, qlength, qsd* return the utilization, throughput, mean queuing time, mean queue length and the standard deviation for the mean queue length, respectively. Procedure *connect* could be used to connect the resource to the appropriate place.

Example:

The following example illustrates the use of the class resource to generate 1 unit of a resource named "CPU" which is then connected to a place referenced by PL.

```
ref(place) PL;
....
new resource("CPU",1).connect(PL);
```

7. Class Place

The class *place* contains everything that is necessary for setting up a place object.

Example:

To create an instance of the class *place* and a reference variable to the object, the following statements are sufficient.

```
ref(place) PL;
....
PL :- new place;
```

8. Class Transition

The class *transition* could be used as a template to declare new transition objects.

Outline:

```
class transition;
   virtual: boolean procedure precond;
            real procedure work;
            ref(bag) procedure outres;
   begin
      ref(subnet) snet;
      ref(bag) inpl, outpl;
      boolean procedure precond;
         ! precond := TRUE if p_i ∈ INPL ⇒ M(p_i)>0,
                      FALSE otherwise;
      real procedure work;
         work := 0;
      ref(bag) procedure outres;
         ! outres :- union of all the output places in outpl ;
      ....
   end;
```

The procedure *work* implements two EPTN transition attributes: *transition time* (z) and *transition procedure* (q). The procedure body implements the transition procedure, while the real value returned by the procedure represents the transition delay. The procedure definitions, given informally and enclosed by "!" and ";", represent the default definitions for the transition attributes. Variable *snet* references the subnet where the transition is located.

Subclass Declaration Syntax:

```
transition class <transition name> (<place list>);
   ref(place) <place list>;
   begin
      [boolean procedure precond;
         precond := <boolean postfix expr>;]
      [real procedure work;
         begin
            <procedure body>;
            work := <transition time>;
         end;]
      [ref(bag) procedure outres;
         outres :- <bag expression>;]

      inpl :- <bag expression>;
      outpl :- <bag expression>;
   end;
```

```
<transition name> ::= <identifier>
<place list> ::= <place> | <place>, <place list>
<place> ::= <identifier>
<boolean postfix expr> ::=
    A(<simple boolean expr>,<simple boolean expr> |
    O(<simple boolean expr>,<simple boolean expr>
<simple boolean expr> ::= M(<place>) <boolean operator> <integer> |
                <boolean postfix expr> | TRUE
<boolean operator> ::= < | > | =
<transition time> ::= <real value expression>
<bag expression> ::=
    U(<simple bag expression>, <simple bag expression>)
<simple bag expression> ::= <place> | NONE | <bag expression>
```

A is a SIMNET built-in boolean function equivalent to a SIMULA AND operator with the side-effect of keeping track of the input places that will be participating in the firing of the transition.

O is a SIMNET built-in boolean function equivalent to the SIMULA boolean operator OR with the side-effect of keeping track of the input places that will be participating in the firing of the transition.

M is a SIMNET built-in function returning the cardinality (marking) of a place.

U is a SIMNET built-in function which returns the union of its two arguments as its value. The result is of type *bag* which is a predefined SIMNET class.

Example:

The following example shows how to use the transition class to declare a transition with the following structure and semantics. We will not be concerned with the interpretations since they are application dependent.

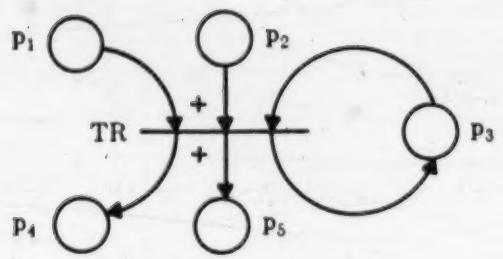

Figure A1.1. An EPTN model of a simple transition.

$pr(TR) = AND(OR(M(p_1)>0, M(p_2)>0), M(p_3)>0)$

$r(TR) = \begin{cases} \{p_4,p_3\} \text{ if COND\_1} \\ \{p_5,p_3\} \text{ otherwise} \end{cases}$

$q(TR) =$ SOME_STATEMENTS

$z(TR) =$ SECS

where COND_1 is a SIMULA boolean expression, SOME_STATEMENTS is some SIMULA statements and SECS is a real value.

```
transition class TR(IP1, IP2, IP3, OP1, OP2, OP3);
   ref(place) IP1, IP2, IP3, OP1, OP2, OP3;
   begin
      boolean procedure precond;
         precond := A(O(M(IP1)>0, M(IP2)>0), M(IP3)>0);
      real procedure work;
         begin
            SOME_STATEMENTS;
            work := SECS;
         end;
      ref(bag) procedure outres;
         outres :- if COND_1 then U(OP2, OP3)
                   then U(OP2, OP3);

      inpl :- U(U(IP1, IP2), IP3);
      outpl :- U(U(OP1, OP2), OP3);
   end;
```

9. Class Subnet

A *subnet* in SIMNET represents an implementation of one part of the net model, but it could also represent the entire system as well. Subnets are declared in terms of places, transitions and other facilities that are required for performance purposes. All the basic mechanisms that are needed to set up a subnet in SIMNET are provided by the class *subnet*.

Subclass Declaration Syntax:

```
subnet class <subnet name> (<place list>);
    ref(place) <place list>;
    begin
        ref(place) <place list>;
        <place generation statements>;
        <transition generation statements>;
        <facility generation statements>;
    end;

<subnet name> ::= <identifier>
<place generation statements> ::= <place> :- new place; |
    <place> :- new place; <place generation statements>
<transition generation statements> ::=
    new <transition name>(<place list>) | new <transition name>
    (<place list>); <transition generation statements>
<facility generation statements> ::= <simple facility generation> |
    <simple facility generation>; <facility generation statements>
<simple facility generation> ::= <source generation statement> |
    <sink generation statement> | <resource generation statement>
<source generation statements> ::= new <source name>.connect(<place>);
<sink generation statements> ::= new sink.connect(<place>);
<resource generation statements> ::= new resource(<nounits>,<nam>).
    connect(<place>);
<nounits> ::= <integer>
<nam> ::= <string>
```

Example:

This example shows how to use the SIMNET class *subnet* to declare a subnet with the following EPTN structure:

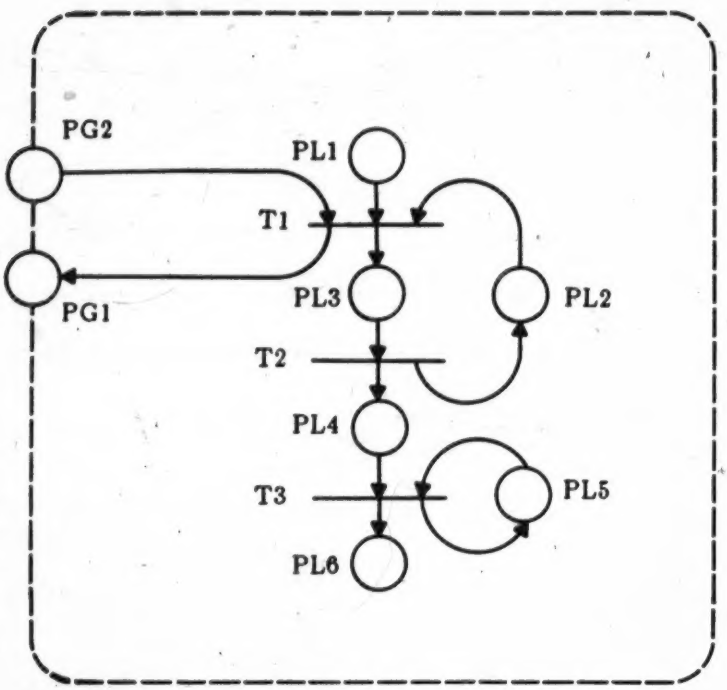

Figure A1.2. An EPTN model of a subnet.

In this example, we are mainly interested in the syntax rather than in the interpretations. We assume the following:

PL1 is connected to source NEWSOURCE.

PL2 is connected to resource CPU.

PL5 is connected to resource IO.

PL6 is connected to sink SNK.

The SIMNET code for the subnet called system is as follows.

```
subnet class system(PG1, PG2);
   ref(place) PG1, PG2;
   begin
      ref(place) PL1, PL2, PL3, PL4, PL4, PL5, PL6;
```

```
    PL1 :- new place; PL2 :- new place;
    PL3 :- new place; PL4 :- new place;
    PL5 :- new place; PL6 :- new place;

    new T1(PG1, PL1, PL2, PG2, PL3);
    new T2(PL3, PL4, PL2);
    new T3(PL4, PL5, PL6, PL5);

    new NEWSOURCE.connect(PL1);
    new resource("CPU", 1).connect(PL2);
    new resource("IO", 1).connect(PL5);
    new sink.connect(PL6);
end;
```

Alternatively, PL1,...,PL6 could be declared as an array of *places* as

```
ref(place) array(1:6);
```

and a **for** loop construct could be used to generate the places.

10. Procedure Setcurenv

Procedure *setcurenv* is a dummy predefined procedure that is invoked by the SIMNET simulator every time a transition is activated. It can be redefined to set up a working environment for the active transition or to do some user-defined functions.

Procedure Redefinition Syntax

```
procedure setcurenv;
    <procedure body>
```

Example:

In the following example, procedure *setcurenv* is redefined so that the reference variable TRANSACTION always refers to current data object, and the reference variable SNET always refers to the current subnet. The attributes of the data object could then be remotely accessed through TRANSACTION within the active transition, and the current subnet could be referred to as SNET.

```
ref(dataset) TRANSACTION;
ref(subnet) SNET;
....
procedure setcurenv;
   begin
      TRANSACTION :- curdata;
      SNET :- cursubnet;
   end;
```

Curdata is a SIMNET global variable that always refers to the data object for the currently active transition, and *cursubnet* is a global variable that always refers to the current subnet.

11. Net Construction

After all the necessary subnets have been declared, the net can then be constructed by generating and linking together the subnets. This is done in the net construction part of the main program.

Net Construction Syntax:

```
   [<global place generation>];
   <subnet generation>;

<global place generation> ::= <place generation statements>
<subnet generation> ::= <simple subnet generation> |
      <simple subnet generation>; <subnet generation>
<simple subnet generation> ::=
      new <subnet name>[(<parameter specification>)]
<parameter specification> ::= <place list>
```

Example:

The following example shows how to construct a net from two predefined subnets S_1 and S_2 linked together by two global places PG_1 and PG_2 as shown in the structure below. The inner structures of the subnets are omitted.

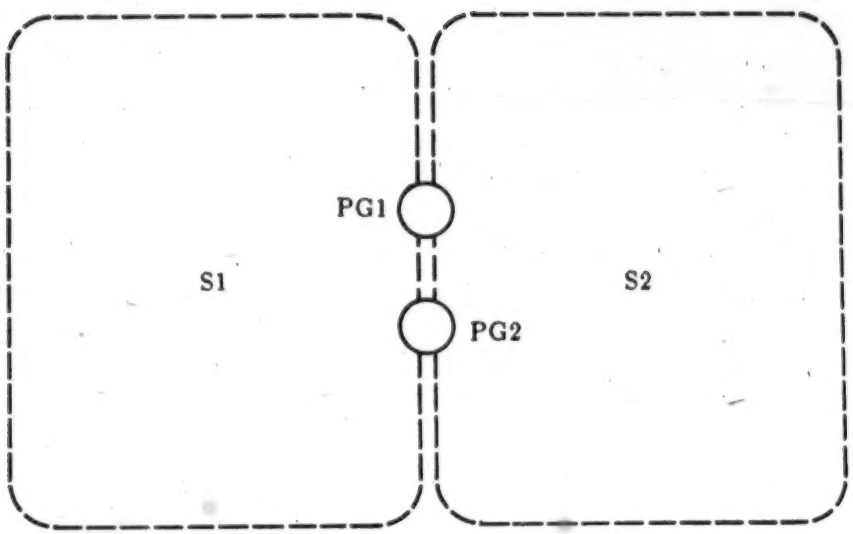

Figure A1.3. An EPTN model of a net.

```
ref(place) PG1, PG2;
 ....
PG1 :- new place;
PG2 :- new place;
new S1(PG1, PG2);
new S2(PG1, PG2);
 ....
```

12. Net Model Simulation

The procedures that are available for (1) debugging a net construction, (2) simulating, tracing and controlling the execution of a net model, and (3) reporting the performance statistics are presented in this section.

12.1. Checking for Errors in the Net Construction

To help the user find bugs in the net construction, two facilities are provided: (1) *check_net* and (2) *display_net*.

Check_net:

The *check_net* procedure could be used to find some major bugs in the net construction. The types of errors that could be reported are listed below, with explanations for those that are not very clear.

1. +++ PLACE PL1 IN SNET 1 - NOT LINKED TO ANY TRANSITIONS
2. +++ PLACE PL4 IN SNET 2 - WILL NEVER BE USED

   This means that the place PL4 in subnet SNET 2 does not have any input transition or source connected to it, so that the place could never receive a token.
3. +++ PLACE PL2 IN SNET 3 - NO INPUT TRANSITION

   This message is usually obtained if a place that is connected to a resource has at least one output transition but no input transition.
4. +++ PLACE PL3 IN SNET 2 - NO OUTPUT TRANSITION

   This means that the place has at least one input transition but no output transition.
5. +++ TRANSITION TR1 IN SNET 3 - NOT LINKED TO ANY INPLACES
6. +++ TRANSITION TR2 IN SNET 4 - TRYING TO REQUEST MORE THAN ONE RESOURCE

   This message is usually obtained if a transition has more than one of its input places connected to a resource.

Display_net:

If the user wants to verify the net structure to see whether the places and transitions are properly linked, the *display_net* procedure could be used. This procedure will output the structure of the net in a format similar to the following:

```
NET STRUCTURE
-------------
PLN = LOCAL PLACE N; PGN = GLOBAL PLACE N
TR IN SNET 1 : IN={PG1,PL1}, OUT={PG2,PL1}
TR IN SNET 2 : IN={PG2,PL1}, OUT={PG1,PL1}
TR IN SNET 3 : IN={PG2,PL1}, OUT={PG1,PL1}
```

The bag of input places to a transition are given by IN={ ... } and the bag of output places by OUT={ ... }.

12.2. Event Tracing

Sometimes it is necessary to trace through the a program step by step to justify the model or to identify obscure errors. SIMNET provides an event tracing facility for this purpose. Event tracing is initially off and could be switched on by a call to the trace routine within the main program. The format for the procedure call is as shown below:

Trace(FROM, TO, UNIT);

where

FROM is an integer value representing the starting point,

TO is an integer value representing the finishing point, and

UNIT is a character value 'T' or 'J' specifying the type of unit to be used for the above values. 'J' represents Jobs while 'T' represents Time.

Examples:

Trace(50, 100, 'T');

prints a trace of all events that occur during the time interval between 50 and 100.

Trace(25, 100, 'J');

prints a trace of all events that occur during the interval after the 25th job enters the system and before the 101st job leaves the system.

If trace is on, a message is sent to the standard output device whenever one of the following events occurs:

1. a new job enters the system from a source.
2. a job leaves the system through a sink.
3. a transaction is enabled.
4. a transaction fires.

As the amount of printing produced by *trace* could be enormous, one should normally restrict the duration of a trace to a minimum. A sample of the output produced by *trace* is shown below.

```
TRACE BEGINS . . .

   TIME          EVENT(S)

  34.5493    SNET 5 - TRANSACTION 16 ARRIVES
             SNET 5 - TR1 IS ENABLED BY TRANSACTION 16
             SNET 5 - TR1 FIRES
             SNET 7 - TR1 IS ENABLED BY TRANSACTION 16
  34.6496    SNET 7 - TR1 FIRES
             SNET 1 - TR2 IS ENABLED BY TRANSACTION 16
  34.6502    SNET 1 - TR2 FIRES
             SNET 1 - TR3 IS ENABLED BY TRANSACTION 16
  34.9502    SNET 1 - TR3 FIRES
             SNET 1 - TR4 IS ENABLED BY TRANSACTION 16
             SNET 1 - TR4 FIRES
             SNET 7 - TR1 IS ENABLED BY TRANSACTION 16
  35.0505    SNET 7 - TR1 FIRES
             SNET 5 - TR5 IS ENABLED BY TRANSACTION 16
             SNET 5 - TR5 FIRES
             SNET 5 - TR6 IS ENABLED BY TRANSACTION 16
  35.0767    SNET 3 - TRANSACTION 17 ARRIVES
             SNET 3 - TR7 IS ENABLED BY TRANSACTION 17
             SNET 3 - TR7 FIRES
             SNET 7 - TR1 IS ENABLED BY TRANSACTION 17
  35.1770    SNET 7 - TR1 FIRES
             SNET 1 - TR3 IS ENABLED BY TRANSACTION 17

TRACE ENDS.
```

12.3. Request for Simulation

After the net for a system has been set up, the model could be simulated by invoking the procedure *simulate*. The format for the procedure call is as follows.

Simulate(DURATION, UNIT);

where

DURATION is an integer value representing the length of the simulation run.

UNIT is a character value 'J' or 'T' representing the unit for the duration.

'J' indicates that the unit for DURATION is Job whereas 'T' indicates that the Time unit is to be used.

Examples:

Simulate(5000, 'T');

would stop the simulation after 5000 time units have elapsed.

Simulate(5000, 'J');

would stop the simulation after 5000 jobs have been processed.

12.4. Clearance of Accumulated Statistics

The *reset* routine could be used to clear all statistics and reset the simulation clock to zero without modifying the status of the model, that is, without altering the tokens in the system. This is usually used to remove the initial bias caused by the transient state of a simulation.

Example:

Reset;

would reset the simulation model.

12.5. Clearance of the Model

The *clear* routine, if invoked, removes all data tokens from the model, reinitializes the resource (simple) tokens to their available places, reset the accumulated statistics, and reset the simulation clock to zero. In brief, clear reinitializes the model as it was when the simulation first started but does not reset the random number generators.

Example: –

Clear;

would reinitialize the simulation model.

12.6. Data Collection Devices

In addition to providing facilities to automatically record the profiles of resources and input sequences, SIMNET also provides data collection devices that could be used whenever necessary to collect statistics for the places and transitions. However, these facilities should be used with care since the cost involved in the data collection could be very high. These facilities are initially off and could be switched on by a call to the procedure *meter_on*. The format for the procedure call is as follows:

Meter_on(NAME);

NAME is the name of the transition or place whose data collection device is to be switched on.

Once the data collection device for a place or transition is on, the statistics will automatically be recorded and could remotely be accessed at the end of the simulation. The statistics that are recorded are as follows:

Places

1. **Integer** *stat.obs* - records the total number of tokens received.
2. **Real** *stat.resetat* - records the time the device was last reset.
3. **Real** *stat.sumt* - records the time integral of the number of tokens residing at the place since *stat.resetat*.
4. **Real** *stat.sumsqt* - records the time integral of the squares of the number of tokens residing at the place since *stat.resetat*.
5. **Real** *stat.min* - records the least number of tokens that have resided at the place, or zero if no tokens has been received.
6. **Real** *stat.max* - records the largest number of tokens that have resided at the place, or zero if no tokens has been received.

Transition:

1. **Integer** *stat.obs* - records the total number of transition firings.
2. **Real** *stat.resetat* - records the time the device was last rest.

12.7. Reporting of Statistics

A standard output of all statistics collected could be produced by a call to the report procedure. A sample of a typical report is shown below.

TIME : 363.02

RESOURCE	USAGE	TPUT	QLENGTH	QTIME
CPU - SNET 1	0.20755	4.20087	0.00000	0.00000
IO - SNET 1	32.29683	2.30841	0.04683	0.02029
CPU - SNET 2	0.14908	3.63564	0.00000	0.00000
IO - SNET 2	10.64622	1.20379	0.00000	0.00000
CPU - SNET 3	0.12787	2.86485	0.00000	0.00000
IO - SNET 3	10.35701	1.18451	0.00275	0.00233
CPU - SNET 4	0.10715	2.75467	0.00000	0.00000
IO - SNET 4	9.99891	1.17073	0.00000	0.00000
CPU - SNET 5	0.12996	2.95300	0.00000	0.00000

```
IO - SNET 5      10.27437   1.19553      0.00000      0.00000
CPU - SNET 6      0.13081   2.97504      0.00000      0.00000
IO - SNET 6      10.26747   1.19828      0.00551      0.00460
NUMBER OF TRANSACTIONS RECEIVED =    200
NUMBER OF TRANSACTIONS COMPLETED =   200
NUMBER OF TRANSACTIONS IN QUEUES:    0
MEAN RESPONSE TIME =          1.31676
VARIANCE =   0.83067
```

However, a user who is not satisfied with either the contents of the output or the way in which the information is presented could write his own report routine using SIMULA. The code for the standard report routine is as follows.

```
procedure report;
 begin
  ref(resource) RES;
  ref(source) SRC;
  ref(sink) SNK;

  integer I, NUMREC, NUMCOM;
  real TMP1, TMP2, SUMRES, SUMSQRES, VARES;
  real MEANRES, CPUTIL, IOUTIL;

  Outtext("TIME : "); Outfix(SIMTIME,2,10); Outimage;
  Outtext("RESOURCE");
  Setpos(23);
  Outtext("USAGE    TPUT        QLENGTH   QTIME");
  Outimage;
  !*** zyqresq.first points to the first resource  ***;
  !*** in the resource queue.                      ***;
  RES :- zyqresq.first;
  while RES =/= none do
   begin
      if RES.meter then begin
        Outtext(RES.nam);
        Outtext(" - ");
        Outtext(RES.snet.nam);
        Outint(RES.snet.sid,3);
        Setpos(20);
        Outfix(RES.usage * 100,5,10);
        Outfix(RES.tput,5,10);
        Outfix(RES.qlength,5,10);
        Outfix(RES.qtime,5,10);
        Outimage;
      end;
      RES :- RES.next;
  end;
  !** zyqsrcq.first points to the first source **;
```

```
!** in the source queue.                    **;
SRC :- zyqsrcq.first;
while SRC =/= none do begin
 NUMREC := NUMREC + SRC.OBS;
 SRC :- SRC.NEXT;
end;
!** zyqsnkq.first points to the first sink   **;
!** in the sink queue.                       **;
SNK :- zyqsnkq.first;
while SNK =/= none do begin
 SUMRES := SUMRES + SNK.sum;
 SUMSQRES := SUMSQRES + SNK.sumsq;
 NUMCOM := NUMCOM + SNK.obs;
 SNK :- SNK.next;
end;
MEANRES := SUMRES/NUMCOM;
if NUMCOM > 1 then
 VARES := (SUMSQRES-(SUMRES**2)/NUMCOM)/(NUMCOM-1)
else
 VARES := 0.0;
Outtext("NUMBER OF TRANSACTIONS RECEIVED = ");
Outint(NUMREC,7); Outimage;
Outtext("NUMBER OF TRANSACTIONS COMPLETED = ");
Outint(NUMCOM,7); Outimage;
Outtext("NUMBER OF TRANSACTIONS IN QUEUES:");
Outint(NUMREC-NUMCOM,5); Outimage;
Outtext("MEAN RESPONSE TIME = ");
Outfix(MEANRES,5,10); Outimage;
Outtext("VARIANCE = ");
Outfix(VARES,5,10); Outimage;
Outimage;
Outimage;
end;
```

13. Utilities

This section describes the utilities that are available in SIMNET. The random seed generation function *newseed* is adapted from [Bir81b]. The random number generators are inherited from the host language SIMULA.

13.1. Well-Spread Seeds

Ten well-spread seeds are available in SIMNET in the global variables: *seed1, seed2, seed3,..., seed10.* Each time SIMNET is set up, the seeds are initialized to the following values:

Seed1 6229297
Seed2 836027
Seed3 3027628
Seed4 2663670
Seed5 4497811
Seed6 668092
Seed7 1370973
Seed8 6575705
Seed9 4953148
Seed10 831231

The seeds could be reset to any values provided by the user or to a randomly chosen seed by a call to the function *newseed*.

An outline of the function newseed, which is adapted from [Bir81a] is listed below. The global integer variables *zyqseed* and *zyqmodulo* are initialized to 907 and 67099547, respectively, when SIMNET is set up.

```
integer zyqseed, zyqmodulo;
....
integer procedure newseed;
   begin
      integer k;
      for k := 7, 13, 15, 27 do
         begin
            zyqseed := zyqseed * k;
            if zyqseed >= zyqmodulo then
               zyqseed := zyqseed mod zyqmodulo;
         end;
      newseed := zyqseed;
   end;
```

Newseed generates well-spread seeds in the following order:

1. 3059270
2. 6107043
3. 3264518
4. 2626272
5. 6502562
6. 1888495
7. 320133

.
.
.

Each seed has its own portion of the basic cycle of length 120633, that is, after 120633 drawings, the underlying r^{th} distribution will start to overlap with $r+1^{st}$ distribution. More information about the seed generator could be obtained from [Bir81a].

13.2. Random Number Generation

There are ten random number generators available in SIMULA that could also be used in SIMNET. These generators require basic drawings from a stream U, and have the side effect of advancing the stream U by one or more values. Below we give the headings and brief descriptions of these functions [DMN84].

1. **Boolean procedure** draw(a,U); **name** U; **real** a; **integer** U;

   The function returns TRUE with probability a, FALSE with probability $1-a$. The value returned is always TRUE if $a \geq 1$, and FALSE if $a \leq 0$.

2. **Integer procedure** randint(a,b,U); **name** U, **integer** a,b,U;

   The value returned is an integer from one of the integers a, $a+1$,..., $b-1$, b, each one with equal probability. The value b must be greater or equal to a.

3. **Real procedure** uniform(a,b,U); **name** U; **real** a,b; **integer** U;

   The function returns values uniformly distributed in the interval $[a,b]$. The value b must be greater than a.

4. **Real procedure** normal(a,b,U); **name** U; **real** a,b; **integer**U U;

   The function returns values normally distributed according to the mean value a and the standard deviation b.

5. **Real procedure** negexp(a,U); **name** U; **real** a; **integer** U;

   The value returned is a drawing from a negative exponential distribution with mean $1/a$.

6. **Integer procedure** poisson(a,U); **name** U; **real** a; **integer** U;

   The value returned is a drawing from a *Poisson* distribution with parameter a.

7. **Real procedure** erlang(a,b,U); **name** U; **value** a,b; **real** a,b; **integer** U;

   The function returns a value drawn from an *Erlang* (*Gamma*) distribution with mean $1/a$ and standard deviation $1/(a\sqrt{b})$. Both a and b must be greater than zero.

8. **Integer procedure** discrete(A,U); **name** U; **array** A; **integer** U;

   Here, A is one-dimensional array of type real, augmented by the element 1 to the right, is interpreted as a step function of the subscript defining a discrete (cumulative) distribution function. The value returned is an integer in the range [*lsb*, *usb*+1], where *lsb* and *usb* are the lower and upper subscript bounds of the array. It is defined as the smallest i such that $A[i]>U$, where U is a basic drawing and $A[usb+1] = 1$.

9. **Real procedure** linear(A,B,U); **name** U; **array** A,B; **integer** U; The function returns a value drawn from a (cumulative) distribution $F_x(x)$, obtained by linear interpolation in a non-equidistant table defined by A and B, such that $A[i]=F_x(B[i])$. It is assumed that (1) A and B are one-dimensional real arrays of the same length, (2) the first and last elements of A are equal to 0 and 1, respectively, and (3) $A[i]\geq A[j]$ and $B[i]\geq B[j]$ for $i>j$.

10. **Integer procedure** histd(A,U); **name** U; **array** A; **integer** U;

    The value returned is an integer in the range [*lsb*, *usb*], where *lsb* and *usb* are the lower and upper subscript bounds of the one-dimensional array A. The latter is interpreted an a histogram defining the relative frequencies of the values.

Appendix 2

SIMNET Implementation of On-Line Information System

This appendix describes the implementation of our On-Line Information system (see Section 4.3.1) in SIMNET.

```
begin  ! Beginning of main program ;
  external class SIMNET=KOOM:SIMNET;

  ! Declaration of input parameters ;
  integer N;
  real MIT, MTT, MTR, T, P, MTP;

  ! Initialization of input parameters ;
  inimage; N := inint;
  inimage; MIT := inreal;
  inimage; MTT := inreal;
  inimage; MTR := inreal;
  inimage; T := inreal;
  inimage; P := inreal;
  inimage; MTP := inreal;

  SIMNET begin
    ref(place) array GP1(1:N), GP2(1:N);
    ref(place) GP3, GP4;
    ref(CUSTOMER) CUST;
    integer I;

    ! Data object declaration ;
    dataset class CUSTOMER;
```

```
    begin
      real TTR, TRR;

      TTR := Negexp(1/MIT, seed1);
      TRR := Negexp(1/MTR, seed2);
    end;

! Source declaration ;
source class SOURCE1;
    begin
      ref(CUSTOMER) procedure newdata;
        newdata :- new CUSTOMER;
      real procedure dist;
        dist := Negexp(1/MIT, seed3);
    end;

! Redefinition of a SIMNET setcurenv procedure ;
procedure setcurenv;
    CUST :- curdata;

! Declaration of Enter Request transition ;
transition class TT1(IP1, IP2, OP1, OP2);
    ref(place) IP1, IP2, OP1, OP2;
    begin
      real procedure work;
        work := CUST.TTR;

      inpl :- U(IP1, IP2);
      outpl :- U(OP1, OP2);
    end;

! Declaration of Read Reply transition ;
transition class TT2(IP1, IP2, OP1, OP2);
    ref(place) IP1, IP2, OP1, OP2;
    begin
      real procedure work;
        work := CUST.TRR;

      inpl :- U(IP1, IP2);
      outpl :- U(OP1, OP2);
    end;

! Declaration of send request from terminal to  ;
! processor transition                          ;
transition class NT1(IP1, IP2, OP1, OP2);
    ref(place) array IP1;
    ref(place) IP2, OP1, OP2;
    begin
      integer I;
      boolean procedure precond;
        begin
          boolean COND;
```

```
            COND := FALSE;
            for I := 1 step 1 until N do
              COND := O(COND, M(IP1(I))>0);
            precond := A(COND, M(IP2)>0);
          end;

        real procedure work;
          work := T;

        inpl :- none;
        for I := 1 step 1 until N do
          inpl :- U(inpl, IP1(I));
        inpl :- U(inpl, IP2);
        outpl :- U(OP1, OP2);
      end;

! Declaration of send reply from processor to  ;
! terminal transition.                         ;
transition class NT2(IP1, IP2, OP1, OP2);
    ref(place) array OP1;
    ref(place) IP1, IP2, OP2;
    begin
      integer I;
      real procedure work;
        work := T;

      ref(bag) procedure outres;
        outres :- U(OP1(CUST.origin.sid), OP2);

      inpl :- U(IP1, IP2);
      outpl :- none;
      for I := 1 step 1 until N do
        outpl :- U(outpl, OP1(I));
      outpl :- U(outpl, OP2);
    end;

! Declaration of Receive Request transition ;
transition class PT1(IP1, IP2, OP1);
    ref(place) IP1, IP2, OP1;
    begin
      real procedure work;
        work := P;

      inpl :- U(IP1, IP2);
      outpl :- U(OP1, none);
    end;

! Declaration of Process Request transition ;
transition class PT2(IP1, OP1);
    ref(place) IP1, OP1;
    begin
      real procedure work;
```

```
        work := Negexp(1/MTP, seed4);

      inpl :- U(IP1, none);
      outpl :- U(OP1, none);
    end;

! Declaration of Send Reply transition ;
transition class PT3(IP1, OP1, OP2);
    ref(place) IP1, OP1, OP2;
    begin
      real procedure work;
        work := P;

      inpl :- U(IP1, none);
      outpl :- U(OP1, OP2);
    end;

! Declaration of terminal site subnet ;
subnet class TERMINAL(IP1, OP1);
    ref(place) IP1, OP1;
    begin
      ref(place) array TP(1:4);
      integer I;

      for I := 1 step 1 until 4 do
        TP(I) :- new place;

      new TT1(TP(1), TP(2), TP(3), OP1);
      new TT2(TP(3), IP1, TP(1), TP(4));

      new SOURCE1.connect(TP(2));
      new sink.connect(TP(4));
      new resource(1, "TERM").connect(TP(1));
    end;

! Declaration of network subnet ;
subnet class NETWORK(IP1, IP2, OP1, OP2);
    ref(place) array IP1, OP2;
    ref(place) IP2, OP1;
    begin
      ref(place) NP1;

      NP1 :- new place;

      new NT1(IP1, NP1, OP1, NP1);
      new NT2(IP2, NP1, OP2, NP1);

      new resource(5, "NET").connect(NP1);
    end;

! Declaration of processor site subnet ;
subnet class PROCESSOR(IP1, OP1);
```

```
        ref(place) IP1, OP1;
        begin
          ref(place) array PP(1:3);
          integer I;

          for I := 1 step 1 until 3 do
              PP(I) :- new place;

          new PT1(IP1, PP(1), PP(2));
          new PT2(PP(2), PP(3));
          new PT3(PP(3), OP1, PP(1));

          new resource(1, "CPU").connect(PP(1));
        end;

      ! NET CONSTRUCTION STATEMENTS ;
      ! creation of global places ;
      for I := 1 step 1 until N do
        begin
          GP1(I) :- new place;
          GP2(I) :- new place;
        end;
      GP3 :- new place;
      GP4 :- new place;
      ! creation of net from subnets and global places ;
      for I := 1 step 1 until N do
        new TERMINAL(GP2(I), GP1(I));
      new NETWORK(GP1, GP4, GP3, GP2);
      new PROCESSOR(GP3, GP4);

      ! SIMULATION CONTROL STATEMENTS ;
      check_net;
      display_net;
      trace(1, 5, 'J');
      simulate(60, 'T');
      report;
    end;
end;
```

Appendix 3

Centralized Locking Algorithm

In this appendix, we give a slightly formal description of the centralized locking algorithm. In the description, we distinguish the central site functions from the non-central site functions since they are quite different. The algorithm is described in a Pascal-like notation.

The concurrency control algorithm executed at all non-central sites is as follows.

```
Procedure Cohortcc;
  Begin
    Wait for an event to occur;
    Case event of
      "new transaction":
        Send lock request for all items in trans. base-set to central site;
      "locks granted": begin
        If (there exists an older transaction which is not in the
              site done-set and the transaction hole-set) then
          Insert transaction in waiting queue
        else begin
          Read transaction base-set;
          Compute new update values for transaction;
          Save new values in log;
          Write "prepare" in log;
          Send intend-to-update message & new update values to all sites;
        end;
      end;
      "acknowledgment": begin
        Increment number of ack's received for transaction by 1;
        If all up sites have acknowledged then begin
```

```
            Write "commit" in log;
            Send commit message to all sites;
            Perform update on local database;
            Write "complete" in log;
            Restart transactions in waiting queue waiting for this transaction
              to complete;
          end;
        end;
        "intend to update": begin
          If transaction can be committed then begin
            Save transaction new update values in stable memory;
            Write "ready" in log;
            Send acknowledgment to originating site;
          end
          else
            Send abort to originating site;
        end;
        "commit": begin
          If (there exists an older transaction which is not in the site
                done-set and the transaction hole-set) then
            Insert transaction in waiting queue;
          else begin
            Write "commit" in log;
            Perform update on local database;
            Insert transaction id. in site done-set;
            Write "complete" in log;
            Restart transactions waiting for this transaction to complete;
          end;
        end;
        "abort":
          Write "abort" in log;
        "central site failure": begin
          Go into failure mode;
          Start election protocol;
        end;
      end
    end;
    {The following algorithm is executed at the central site only.}

Procedure centralcc;
  Begin
    Wait for an event to occur;
    Case event of
      "new transaction":
        Request locks for items in transaction base-set locally;
      "lock request": begin
        While (transaction has not received all its locks) do begin
          If next transaction item is already locked then
            Insert transaction in item queue;
          else
            lock item for transaction;
```

```
        end;
        Insert transaction in site hole-set;
        If transaction is local then
          Read transaction base-set;
          Compute new update values;
          Save New update values in log;
          Write "prepare" in log;
          Send intend-to-update message & new update values to all sites;
        end
        else
          Send locks-granted message and hole-set to originating site;
      end;
      "intend to update": begin
        If transaction could be committed then begin
          Save new update values in log;
          Write "ready" in log;
          Send acknowledgment to originating site;
        end
        else
          Send abort message to originating site;
        end;
      "acknowledgment": begin
        Increment number of ack's received for transaction by 1;
        If all up sites have agreed to commit then begin
          Write "commit" in log;
          Send commit message to all sites;
          Perform update on local database;
          write "complete" in log;
          Insert transaction id. in site done-set;
          Release all locks held by transaction;
          Restart transactions waiting for this transaction to complete;
          Restart transactions in item queues waiting for locks held by
            transaction;
        end
      end;
      "abort": begin
        Write "abort" in log;
        Send abort to all sites;
      end;
      "coordinator failure":
        Start termination protocol;
    end;
  end;
```

Appendix 4

Distributed Locking Algorithm

This appendix contains a formal description of the distributed locking concurrency control algorithm. The algorithm executed at every site is as follows.

```
Procedure allcc;
  Begin
    Wait for an event to occur;
    Case event of
     "new transaction": begin
        Request locks for all items in transaction write-set from all sites;
        Request locks for all items in transaction base-set locally;
      end;
     "lock request": begin
        While (transaction has not obtained all its locks) do begin
          If next transaction item is already locked  AND conflicting
              transaction is younger than transaction requesting lock AND
              is not in a ready-to-commit OR commit state then
            Insert transaction in item queue in ascending order of timestamp;
          else
            Lock item for transaction;
          end;
          If transaction is local then
            Read transaction read-set;
            Compute new update values;
            Write "prepare" in log;
            Save new update values in log;
            Send intend-to-update message and new update values to all sites;
          end
```

```
          else
            Send locks-granted to transaction originating site;
        end;
      "intend to update": begin
        If transaction can be committed then
          Write "ready" in log;
          Save transaction new update values in log;
          send acknowledgment to originating site;
        end
        else
          Send abort to originating site;
        end;
      "acknowledgment": begin
        Increment number of ack's received for transaction by 1;
        If all up sites have agreed to commit then begin
          Write "commit" in log;
          Send commit to all sites;
          Perform update on local database;
          Write "complete" in log;
          Release all locks held by transaction;
          Restart transactions in item queues waiting for locks held by
           transaction;
        end;
       end;
      "abort": begin
        Write "abort" in log;
        Release locks held by transaction;
        Restart transactions waiting for locks held by this transaction;
       end;
      "coordinator failure":
        Elect new coordinator to terminate unfinished transactions;
    end;
  end;
end.
```

Appendix 5

Performance Results

The performance results obtained for the CL and DL algorithms are listed in this appendix.

The results were obtained from simulation runs of about 10,000 transactions each after the system has reached its steady state. To give an accuracy of the results, 95% confidence intervals are computed for the mean response times $\overline{R}$ as:

$$\left(\overline{R} - 1.96\sqrt{\frac{s^2}{n}}, \overline{R} + 1.96\sqrt{\frac{s^2}{n}}\right)$$

where s^2 is the sample variance obtained from the simulation and n is the total number of samples. The above equation assumes that the n samples are independent. In reality, the samples obtained from simulation runs are usually correlated. That is, the samples are not quite independent since the value of one sample can affect the values of some other samples. However, if the number of samples is large, which is quite true in our case, we can assume that the samples are independent and obtain satisfactory results ([Gor78] and [Gar79]).

Table A5.1. Effect of interarrival time on response time.

Interarrival Time	CL		DL	
	Response Time	Confidence Interval	Response Time	Confidence Interval
4	5.332	0.680	3.518	0.049
5	2.350	0.024	2.451	0.032
6	1.807	0.017	2.055	0.026
7	1.590	0.012	1.723	0.021
8	1.465	0.012	1.723	0.021
9	1.389	0.011	1.635	0.020
10	1.335	0.010	1.570	0.019
11	1.298	0.009	1.515	0.018
12	1.270	0.009	1.476	0.017
13	1.247	0.009	1.445	0.017
14	1.228	0.009	1.420	0.016
15	1.213	0.008	1.400	0.016
16	1.203	0.008	1.382	0.016
17	1.192	0.008	1.365	0.016
18	1.185	0.008	1.352	0.015
19	1.177	0.008	1.340	0.016
20	1.170	0.008	1.331	0.015

Table A5.2. Effect of mean interarrival time on I/O utilization.

Interarrival Time	CL		DL
	I/O Utiliz. at Central Site	I/O Utiliz. at Other Sites	I/O Utiliz. at All Sites
4	87.445	31.331	73.100
5	69.965	25.070	58.322
6	58.305	20.893	48.555
7	49.979	17.909	41.608
8	43.735	15.671	36.402
9	38.875	13.928	32.362
10	34.988	12.536	29.113
11	31.808	11.396	26.466
12	29.157	10.447	24.256
13	26.915	9.643	22.388
14	24.992	8.955	20.791
15	23.326	8.358	19.402
16	21.868	7.835	18.188
17	20.582	7.374	17.118
18	19.438	6.965	16.167
19	18.415	6.598	15.316
20	17.494	6.268	14.551

Table A5.3. Effect of mean interarrival time on CPU utilization.

Interarrival Time	CL		DL
	CPU Utiliz. at Central Site	CPU Utiliz. at Other Sites	CPU Utiliz. at All Sites
4	0.584	0.452	0.468
5	0.392	0.324	0.374
6	0.316	0.264	0.311
7	0.268	0.224	0.267
8	0.233	0.195	0.233
9	0.206	0.173	0.207
10	0.185	0.156	0.187
11	0.168	0.141	0.170
12	0.154	0.129	0.155
13	0.142	0.119	0.143
14	0.132	0.111	0.133
15	0.123	0.103	0.124
16	0.115	0.097	0.117
17	0.109	0.091	0.111
18	0.103	0.086	0.104
19	0.097	0.081	0.098
20	0.092	0.077	0.093

Table A5.4. Effect of mean interarrival time on mean number of messages.

Interarrival Time	CL	DL
	Mean Number of Messages	Mean Number of Messages
4	16.674	25.099
5	16.672	25.048
6	16.670	25.022
7	16.671	25.015
8	16.669	25.019
9	16.669	25.018
10	16.669	25.013
11	16.669	25.012
12	16.669	25.003
13	16.669	25.001
14	16.669	25.001
15	16.669	25.001
16	16.669	25.000
17	16.669	25.000
18	16.669	25.000
19	16.669	25.000
20	16.669	25.000

Table A5.5. Effect of mean base-set size on mean response time.

Base-set Size	CL		DL	
	Response Time	Confidence Interval	Response Time	Confidence Interval
1	0.642	0.002	0.712	0.002
2	0.771	0.003	0.864	0.007
3	0.926	0.005	1.059	0.009
4	1.112	0.007	1.290	0.014
5	1.335	0.010	1.570	0.019
6	1.594	0.013	1.849	0.021
7	1.929	0.018	2.283	0.031
8	2.341	0.024	2.747	0.040
9	2.869	0.032	3.301	0.051
10	3.579	0.043	3.966	0.060
11	4.591	0.060	4.763	0.070
12	6.113	0.086	5.697	0.071
13	9.001	0.121	6.995	0.080
14	.....	.....	8.525	0.092
15	.....	.....	10.736	0.130

Table A5.6. Effect of mean base-set size on I/O utilization.

Base-set Size	CL		DL
	I/O Utiliz. at Central Site	I/O Utiliz. at Other Sites	I/O Utiliz. at All Sites
1	11.637	5.582	11.492
2	17.166	7.278	15.855
3	23.130	9.017	20.305
4	29.130	10.774	24.834
5	34.988	12.536	29.348
6	40.963	14.295	33.863
7	46.954	16.058	38.340
8	52.948	17.824	42.855
9	58.944	19.580	47.364
10	64.959	21.343	51.931
11	70.978	23.109	56.453
12	76.957	24.857	60.991
13	.....	.....	65.620
14	.....	.....	70.165
15	.....	.....	74.662

Table A5.7. Effect of number of sites on mean response time.

Number of Sites	CL		DL	
	Response Time	Confidence Interval	Response Time	Confidence Interval
2	1.164	0.008	1.271	0.014
4	1.245	0.009	1.411	0.017
6	1.339	0.010	1.570	0.019
8	1.477	0.011	1.773	0.021
10	1.676	0.016	2.054	0.026
11	1.832	0.020	2.196	0.030
12	2.040	0.025	2.385	0.035
13	2.341	0.030	2.830	0.040
14	2.878	0.035	3.225	0.046
15	3.776	0.042	3.225	0.051

Table A5.8. Effect of number of sites on I/O utilization.

Number of Sites	CL		DL
	I/O Utiliz. at Central Site	I/O Utiliz. at Other Sites	I/O Utiliz. at All Sites
2	12.877	5.101	11.465
4	24.804	8.989	20.465
6	36.304	12.860	29.348
8	48.945	16.943	38.845
10	60.817	20.891	48.057
11	65.922	22.617	52.129
12	71.400	24.409	56.451
13	77.143	26.261	60.818
14	82.360	27.961	65.068
15	88.150	29.880	69.705

Table A5.9. Effect of transmission time on mean response time.

Message Trans. Time	CL		DL	
	Response Time	Confidence Interval	Response Time	Confidence Interval
0.05	1.135	0.009	1.389	0.019
0.10	1.335	0.010	1.570	0.019
0.15	1.450	0.010	1.760	0.019
0.20	1.600	0.011	1.954	0.018
0.25	1.750	0.012	2.153	0.018

Table A5.10. Effect of I/O synchronization time on mean response time.

I/O Synchroni. Time	CL		DL	
	Response Time	Confidence Interval	Response Time	Confidence Interval
0.00	0.739	0.003	0.006	0.010
0.01	0.934	0.005	1.037	0.010
0.02	1.335	0.010	1.570	0.019
0.025	1.335	0.010	1.570	0.019
0.03	1.508	0.012	1.805	0.023
0.04	1.948	0.017	2.362	0.032
0.05	2.550	0.026	3.089	0.043

Table A5.11. Effect of I/O synchronization time on I/O utilization.

I/O Synchroni. Time	CL		DL
	I/O Utiliz. at Central Site	I/O Utiliz. at Other Sites	I/O Utiliz. at All Sites
0.00	11.466	11.469	11.747
0.01	20.973	11.896	18.719
0.02	30.465	12.323	25.733
0.025	34.988	12.536	29.113
0.03	39.897	12.749	32.653
0.04	49.335	13.177	39.561
0.05	58.759	13.603	46.920

Table A5.12. Effect of interarrival time on mean response time ($IO_s = 0$).

Interarrival Time	CL		DL	
	Response Time	Confidence Interval	Response Time	Confidence Interval
3	0.883	0.870	1.015	0.047
4	0.829	0.670	0.928	0.045
5	0.798	0.014	0.882	0.030
6	0.781	0.016	0.853	0.024
7	0.769	0.011	0.833	0.020
8	0.759	0.010	0.821	0.019
10	0.747	0.008	0.803	0.018
12	0.738	0.007	0.791	0.016
14	0.733	0.006	0.783	0.014
16	0.730	0.005	0.777	0.013
20	0.722	0.003	0.769	0.011

Table A5.13. Effect of mean interarrival time on I/O utilization ($IO_s = 0$).

Interarrival Time	CL	DL
	I/O Utiliz. at All Sites	I/O Utiliz. at All Sites
3	38.993	39.286
4	29.357	29.453
5	23.529	23.576
6	19.808	19.638
7	16.806	16.826
8	14.723	14.729
10	11.781	11.783
12	9.820	9.810
14	8.412	8.417
16	7.362	7.365
20	5.893	5.889

Table A5.14. Effect of database size on mean response time.

Interarrival Time	CL		DL	
	Response Time	Confidence Interval	Response Time	Confidence Interval
25	2.100	0.018	2.171	0.076
50	1.527	0.014	1.877	0.063
100	1.441	0.012	1.682	0.056
150	1.407	0.012	1.643	0.045
200	1.391	0.011	1.631	0.041
250	1.376	0.011	1.618	0.035
300	1.368	0.011	1.601	0.030
400	1.356	0.010	1.595	0.027
500	1.349	0.010	1.580	0.026
600	1.342	0.010	1.573	0.024
700	1.339	0.010	1.569	0.021
800	1.337	0.010	1.568	0.019
900	1.335	0.010	1.566	0.019
1000	1.335	0.010	1.565	0.019
1100	1.333	0.010	1.563	0.019
1200	1.333	0.010	1.562	0.019
1300	1.332	0.010	1.561	0.019
1400	1.329	0.010	1.560	0.019

Table A5.15. Effect of interarrival time on response time for different mean time between failures.

MTBF	Interarrival Time	CL		DL	
		Response Time	Confidence Interval	Response Time	Confidence Interval
5	4	2.938	0.214	2.775	0.059
	7	1.432	0.006	1.696	0.006
	10	1.285	0.006	1.469	0.006
	15	1.206	0.006	1.369	0.006
	20	1.178	0.006	1.317	0.006
15	4	3.346	0.216	3.047	0.019
	7	1.458	0.006	1.744	0.006
	10	1.298	0.006	1.501	0.006
	15	1.214	0.006	1.377	0.006
	20	1.830	0.006	1.315	0.006
37	4	5.332	0.680	3.518	0.049
	7	1.590	0.012	1.723	0.021
	10	1.335	0.010	1.570	0.019
	15	1.213	0.008	1.400	0.016
	20	1.170	0.008	1.331	0.015

Table A5.16. Effect of mean interarrival time on I/O utilization for different mean time between failures.

MTBF	Interarrival Time	CL	DL
		Mean I/O Utiliz.	Mean I/O Utiliz.
5	4	53.426	63.712
	7	32.808	36.457
	10	23.085	25.553
	15	15.577	17.241
	20	11.800	13.075
15	4	56.681	67.172
	7	33.069	38.391
	10	23.280	26.652
	15	15.641	18.112
	20	11.859	13.726
37	4	59.388	73.100
	7	33.944	41.608
	10	23.762	29.113
	15	15.842	19.402
	20	11.881	14.551